Beginning Django 3 Development

Build Full Stack Python Web Applications

Greg Lim

Beginning Django 3 Development

Build Full Stack Python Web Applic

Beginning Django 3: Build Full Stack Python Web Applications

Greg Lim - Daniel Correa

First Edition: November 2021

Co-Author: Daniel Correa

Table of Contents

PREFACE

About this book

In this book, we take you on a fun, hands-on and pragmatic journey to learning Django full stack development. You'll start building your first Django app within minutes. Every chapter is written in a bite-sized manner and straight to the point as we don’t want to waste your time (and most certainly ours) on the content you don't need. In the end, you will have the skills to create a Movies review app and deploy it to the Internet.

In the course of this book, we will cover:

- Chapter 1: Introduction
- Chapter 2: Installing Python and Django
- Chapter 3: Understanding the Project Structure
- Chapter 4: Creating Our First App
- Chapter 5: URLs
- Chapter 6: Generating HTML Pages with Templates
- Chapter 7: Adding Bootstrap to Our Site
- Chapter 8: Adding a Search Form
- Chapter 9: Models
- Chapter 10: Django Admin Interface
- Chapter 11: Displaying Objects from Admin
- Chapter 12: Revisiting Concepts - Adding A News App
- Chapter 13: Understanding the Database
- Chapter 14: Displaying Objects in Admin
- Chapter 15: Extending Base Templates
- Chapter 16: Static Files
- Chapter 17: Movie Detail Page
- Chapter 18: Creating a Signup Form
- Chapter 19: Creating a User
- Chapter 20: Showing if a User is Logged In
- Chapter 21: Logout
- Chapter 22: Log In
- Chapter 23: Letting Users Post Movie Reviews
- Chapter 24: Creating a Review
- Chapter 25: Listing Reviews
- Chapter 26: Updating a Review
- Chapter 27: Deleting a Review
- Chapter 28: Authorization
- Chapter 29: Deployment

The goal of this book is to teach you Django development in a manageable way without overwhelming you. We focus only on the essentials and cover the material in a hands-on practice manner for you to code along.

Working Through This Book

This book is purposely broken down into short chapters where the development process of each chapter will center on different essential topics. The book takes a practical hands on approach to learning through practice. You learn best when you code along with the examples in the book.

Requirements

No previous knowledge on Django or Python is required, but you should have basic programming knowledge. We will explain concepts that are difficult to understand as we move along.

Getting Book Updates

To receive updated versions of the book, subscribe to our mailing list by sending a mail to support@i-ducate.com . I try to update my books to use the latest version of software, libraries and will update the codes/content in this book. So, do subscribe to my list to receive updated copies!

Code Examples

While it is best that you type all the code by hand yourself, if you are stuck with a coding example or meet a strange error, you can obtain the source code of the completed project at www.greglim.co/p/django and check your code against it.

Chapter 1: Introduction

Welcome to Beginning Django 3! This book focuses on the key tasks and concepts to get you started to learn and build Django applications in a faster pace. It is designed for readers who don't need all the details about Django at this point in the learning curve but concentrate on what you really need to know.

So what's Django? Django is a free, open source web framework for building modern Python web applications. Django helps you quickly build web apps by abstracting away many of the repetitive challenges in building a website, such as:

- connecting to a database,
- handling security
- user authentication
- URL routes,
- display content on a page through templates and forms
- supporting multiple database backends
- admin interface and more.

This allows developers to focus on building a web application's functionality rather than reinvent the wheel for standard web application functions.

Django is one of the most popular frameworks available and is used by established companies like Instagram, Pinterest, Mozilla, National Geographic and more. Yet, it is easy enough to be used for beginning startups and personal projects.
There are other popular frameworks like Flash in Python and Express in JavaScript (Greg has written a book about Express). But they provide the minimum required for a simple web page and developers have to do more foundational work, like installing and configuring third-party packages on their own for basic website functionality.

The App We Will Be Building

We will build a Movie reviews app which lets users view and search for movies. They can also log in and post reviews on the movies (fig. 1.1, 1.2, 1.3).

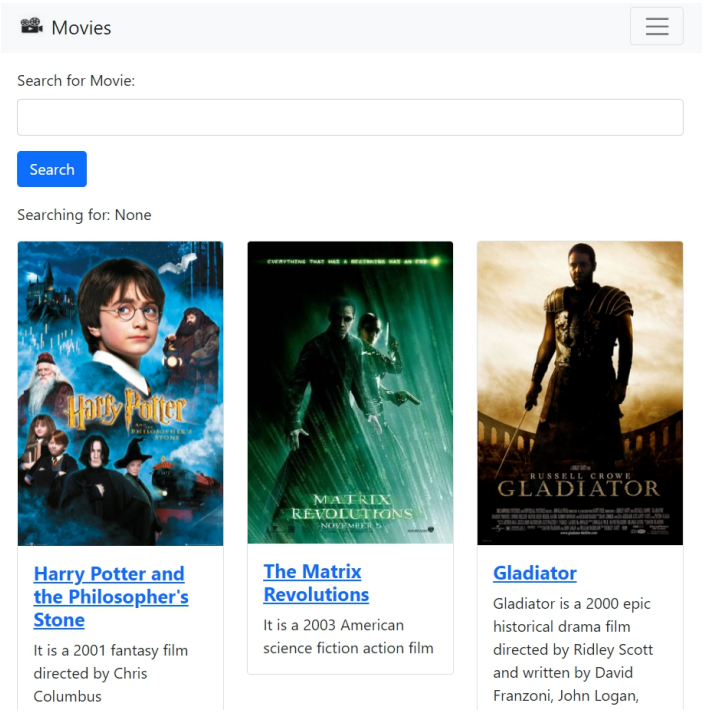


Figure 1.1 – Home Page with search functionality

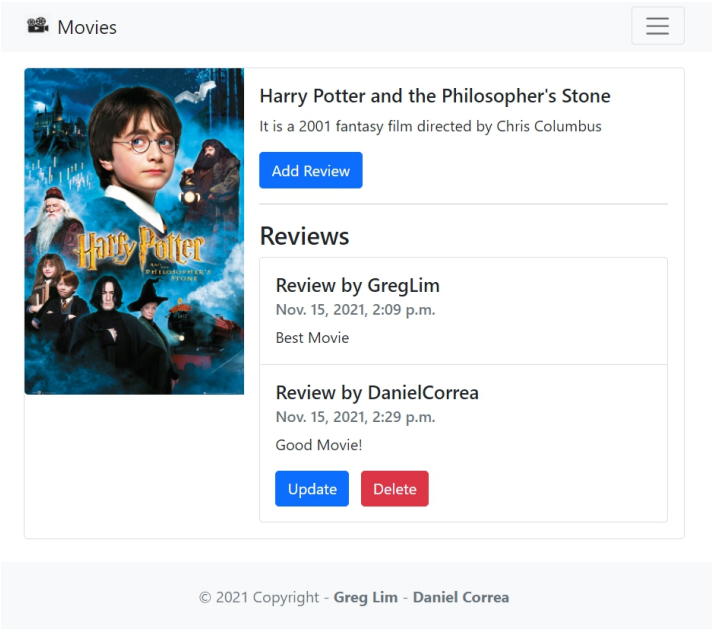


Figure 1.2 – Movie page listing reviews

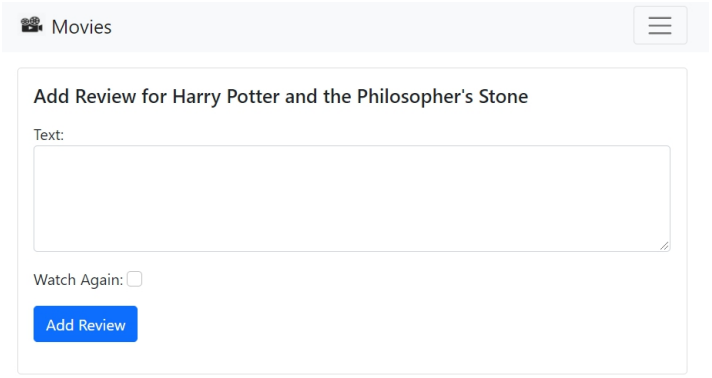


Figure 1.3 – Create Review

Users can see the list of reviews in a Movie ' s page and post/edit/delete their own review if they are logged in. They will not be able edit/delete other ' s reviews though. Through this app, we will learn a lot of concepts like forms, user authorization, permissions, foreign keys and more. By the end of this book, you will be confident creating your own Django projects.

We will begin by installing Python and Django in the next chapter.

Chapter 2: Installing Python and Django

Installing Python

Let ' s check if we have Python installed and if so, what version is it.

If you are using a Mac, open your Terminal. If you are using Windows, open your Command Prompt. For convenience, I will address both the Terminal and Command Prompt as ' Terminal ' throughout the book.

We will need to check if we have at least Python 3.6 in order to use Django 3. To do so, go to your Terminal and run:

```
python3 --version
```

(or *python* on Windows)

This shows the version of Python you installed. Make sure that the version is at least 3.6. If it is not so, get the latest version of Python by going to *python.org* . Go to ' Downloads ' and install the version for your OS.

After the installation, run `python3 --version` again and it should reflect the latest version of Python e.g. Python 3.9.7 (at time of book ' s writing)

Installing Django

We will be using *pip* to install Django. *pip* is the standard package manager for Python to install and manage packages not part of the standard Python library.

First check if you have *pip* installed by going to the Terminal and running:

```
pip3
```

(or *pip* on Windows)

If you have *pip* installed, it should display a list of *pip* commands.

To install Django, run the command:

```
pip3 install django
```

This will retrieve the latest Django code and install it in your machine. After installation, close and re-open your Terminal.

Ensure you have installed Django by running:

```
python3 -m django
```

It will show you all the options you can use (fig. 2.1):

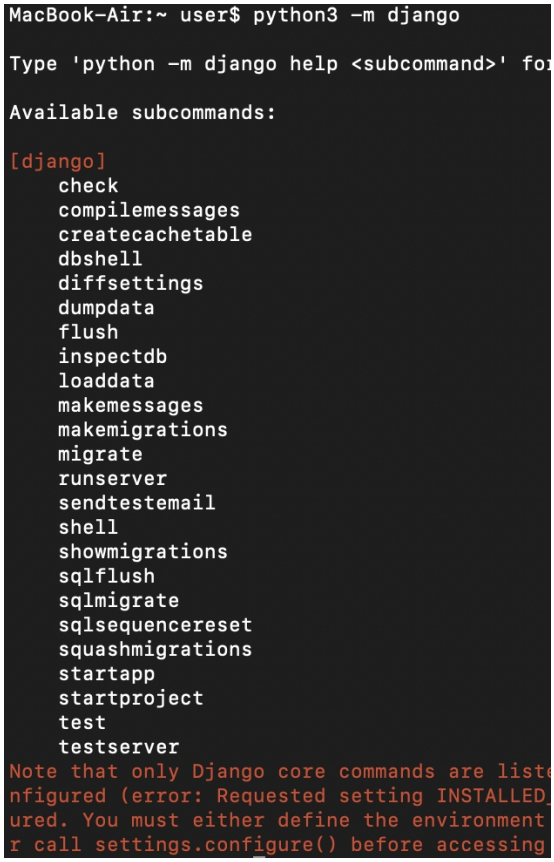


Figure 2.1

Along the course of the book, you will progressively be introduced to some of the options. For now, we will use the *startproject* option to create a new project.

In Terminal, navigate to a folder on your computer where you want to store your Django project e.g. Desktop. In that folder, run:

```
python3 -m django startproject <project_name>
```

In our case, say we want to name our project ‘ moviereviews ’ . We run:

```
python3 -m django startproject moviereviews
```

A ‘ moviereviews ’ folder will be created. We will discuss its contents later. For now, let ’ s run our first website on the Django local web server.

Running the Django Local Web Server

In Terminal, *cd* to the created folder:

```
cd moviereviews
```

and run:

```
python3 manage.py runserver
```

When you do so, you start the local web server on your machine (for local development purposes). There will be a URL link http://127.0.0.1:8000/ (equivalent to http://localhost:8000). Open the link in a browser and you will see the default landing page (fig. 2.2):

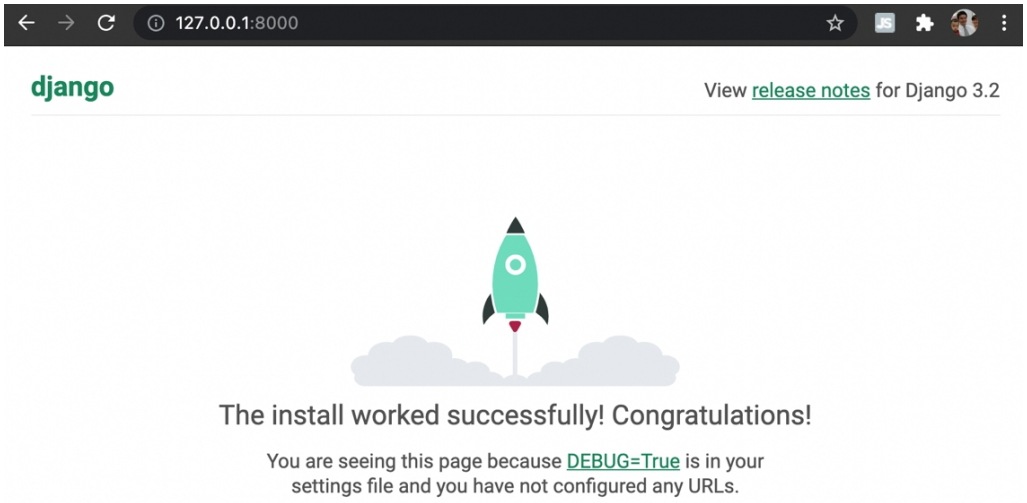


Figure 2.2

This means that your local webserver is running and serving the landing page! To stop the local server, type Control+c in the Terminal.

In the next chapter, we will look inside the project folder that Django has created for us and understand it better.

Chapter 3: Understanding the Project Structure

Let ' s look at the project files that have been created for us. Open the project folder *moviereviews* in a code editor (I will be using the VSCode editor in this book - https://code.visualstudio.com/).

manage.py

You will see a file *manage.py* which we should not tinker (fig. 3.1).

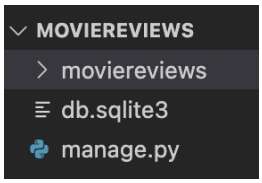


Figure 3.1

manage.py helps us do administrative things. For example, we earlier ran:

```
python3 manage.py runserver
```

to start the local web server. We will later illustrate more of its administrative functions e.g. creating a new app – *python3 manage.py startapp*

db.sqlite3

We also have the *db.sqlite3* file that contains our database. We touch more on this file in the *Models* chapter.

moviereviews folder

You will find another folder of the same name *moviereviews* . To avoid confusion and distinguish between the two *moviereviews* folder, we will keep the inner *moviereviews* folder as it is, and rename the outer folder to *moviereviewsproject* (fig. 3.2).

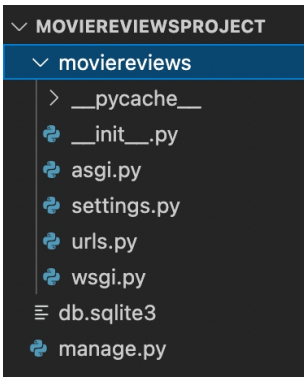


Figure 3.2

In it, we have a *__pycache__* folder that stores compiled bytecode when we generate our project. You can largely ignore this folder. Its purpose is to make your project start a little faster by caching the compiled code that it can readily be executed.

__init__.py specifies what to run when Django launches for the first time. Again, we can ignore this file.

asgi.py allows for an optional Asynchronous Server Gateway Interface to run. *wsgi.py* which stands for Web Server Gateway Interface helps Django serve our web pages. Both files are used when deploying our app. We will revisit them later when we deploy our app.

urls.py

This is the file which tells Django which pages to render in response to a browser or URL request. For example, when someone enters a url http://localhost:8000/123 , the request comes into *urls.py* and gets routed to a page based on the paths specified there. We will later add paths to this file and better understand how it works.

settings.py

The *settings.py* file is an important file controlling our project ' s settings. It contains several properties:

- BASE_DIR determines where on your machine the project is situated at

- SECRET_KEY is used when you have data flowing in and out of your website. Do not ever share this with others
- DEBUG – Our site can run in debug or not. In debug mode, we get detailed information on errors. For e.g. if I try to run http://localhost:8000/123 in the browser, I will see a 404 page not found with details on the error (fig. 3).

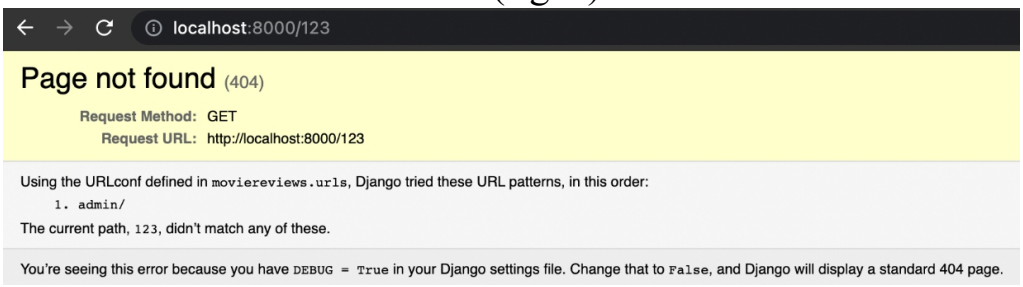


Figure 3

If DEBUG = False, we will see a generic 404 page without error details. While developing our project, we set DEBUG = True to help us with debugging. When deploying our app to production, we should set DEBUG to False.

- INSTALLED_APPS allow us to bring in different pieces of code into our project. We will see this in action later.
- MIDDLEWARE are built-in Django functions to help in our project
- ROOT_URLCONF specify where our urls are
- TEMPLATES help us return HTML code
- AUTH_PASSWORD_VALIDATORS allow us to specify validations we want on passwords. E.g. a minimum length

There are some other properties in *settings.py* like LANGUAGE_CODE and TIME_ZONE but we have focused on the more important ones. We will later revisit this file and see how relevant it is in developing our site. Let ' s next create our first app!

Chapter 4: Creating Our First App

In chapter two, we created a Django project. A single Django project can contain one or more *apps* that work together to power a web application. Django uses the concept of projects and apps to keep code clean and readable.
For example, in a Movies review site such as *rottentomatoes.com* (fig. 4.1), we can have an app for listing movies, an app for listing news, an app for payments, an app for user authentication etc.

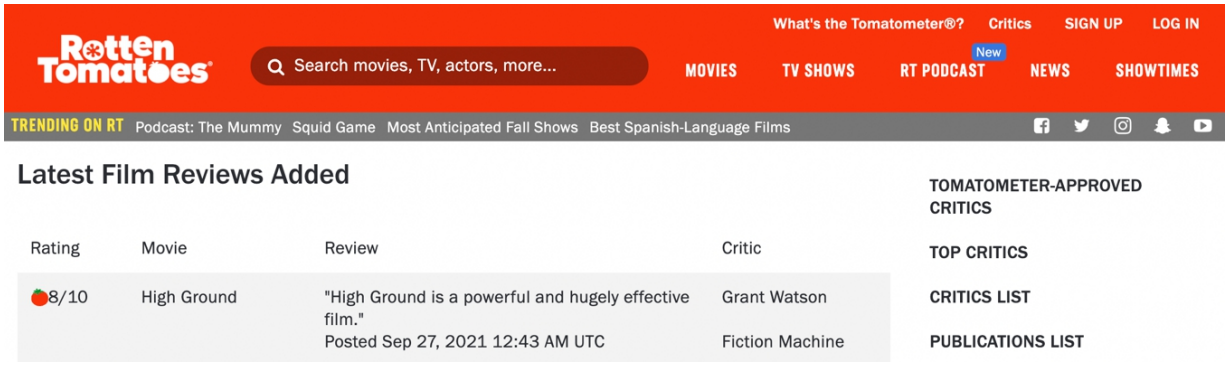


Figure 4.1

Apps in Django are like pieces of a website. You can create an entire website with one single app. But it is useful to break it up into different apps each representing a clear function.

Our movies review site will begin with one app. We will later add more as we progress. To add an app, in the Terminal, stop the server (Cmd + c). Navigate to the *moviereviewsproject* folder (**not *moviereviews***), and run in Terminal:

```
python3 manage.py startapp <name of app>
```

In our case, we will add a *movie* app. So run:

```
python3 manage.py startapp movie
```

A new folder *movie* will be added to the project. As we progress along the

book, we will explain the files that are inside the folder.

Though our new app exists in our Django project, Django doesn ' t ' know ' it till we explicitly add it. To do so, we specify it in *settings.py* . So go to */moviereviews/settings.py* , under INSTALLED_APPS, you will see six built-in apps already there. Add the app name (shown in **bold**):

```
…
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
     'movie',
]
…
```

Back in the Terminal, run the server with:

```
python3 manage.py runserver
```

The server should run without issues. We will learn more about apps in the course of the book.

Currently, you may notice a message in the Terminal when you run the server:

" You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them . "

We will see how to address it later. But for now, remember that we can have one or more apps inside a project.

CHAPTER 5: URLs

Currently, we just have a default landing page provided by Django. How do we create our own custom pages and have different urls to route to them?

Remember that */moviereviews/urls.py* is referenced each time someone types in a url on our website. e.g. *localhost:8000/hello* .

For now, we get an error page when we go to the above url. So, how do we display a proper page for it? Each time a user types in a url, the request passes through *urls.py* and see if the url matches any defined paths so that the Django server can return an appropriate response.

urls.py currently has the code:

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

When a request passes through *urls.py* , it will try to match a *path* object in *urlpatterns* . For e.g. if a user enters http://localhost:8000/admin into the browser, it will match the path ‘ admin/ ’ . The server will then respond with the Django admin site (fig. 5.1 - we will explore this later).

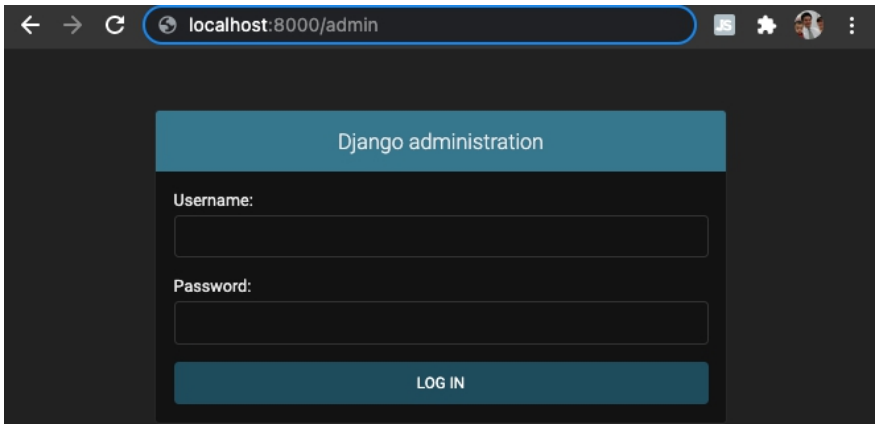


Figure 5.1

localhost:8000/hello however returns a 404 not found page because there isn ' t any matching paths.

To illustrate creating a custom path, let ' s create a path for a home page. Add the below codes in **bold** into *urls.py* :

```
from django.contrib import admin
from django.urls import path
from movie import views as movieViews

urlpatterns = [
    path('admin/', admin.site.urls),
     path('', movieViews.home),
]
```

Code Explanation

```
urlpatterns = [
    path('admin/', admin.site.urls),
     path('', movieViews.home),
]
```

We added a new path object with the path ' ' . That is, it matches the url ' localhost:8000/ ' for a home page. If there is such a match, we return *movieViews.home* which is a function that returns the home page view.

```
from movie import views as movieViews
```

Where do we get *movieViews.home* from? We import it from */movie/views.py* . Note that it is not */* ***moviereviews*** */views.py.* The views are stored in the individual apps themselves i.e. */movie* . Because we have not defined the *home* function in *views.py* , let ' s proceed to do so.

/movie/views.py

In *views.py* , add the following in **bold** :

```
from django.shortcuts import render
from django.http import HttpResponse
```

```
def home(request):
    return HttpResponse('<h1>Welcome to Home Page</h1>')
```

We created a function *home* which returns a HTML markup in a *HTTPResponse* . We imported the built-in *HttpResponse* method to return a response object to the user. Save the file and if you go back to http://localhost:8000 , you should see the home page displayed (fig. 5.2):

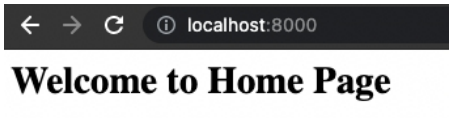


Figure 5.2

Congratulations! We have added a new home path ‘ localhost:8000/ ’ that returns a home page. Now, let ’ s try to create another path for an ‘ About ’ page when a user navigates to *localhost:8000/about* .

In */moviereviews/urls.py* , add the path in **bold** :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home),
    path('about/', movieViews.about),
]
```

So if a url matches the ‘ about/ ’ path, it will return the *about* function. Let ’ s create the *about* function in */movie/views.py* :

```
…
def home(request):
    return HttpResponse('<h1>Welcome to Home Page</h1>')

def about(request):
    return HttpResponse('<h1>Welcome to About Page</h1>')
```

Save the file and when you navigate to *localhost:8000/about* , it will show the About page.

*Notice that when we make changes to a file and save it, Django watches the file changes and reloads the server with the changes. So we don ' t have to manually restart the server each time there is a code change.

We now know how to create custom paths and return views. Notice that the *urls.py* is located in the project ' s main folder *moviereviews* . All requests to the site will go through *urls.py* . But for views, e.g. Home and About views, they are located in the individual app folders e.g. *movie/views.py* . This allows us to separate views according to the app they belong to.

So far, we are just returning simple HTML markups. What if we want to return full HTML pages? We can return them as what we are doing now. But it will be more ideal if we can define the HTML page in a separate file of its own. Le t ' s see how to do so in the next chapter.

Chapter 6: Generating HTML Pages with Templates

Every web framework needs a way to generate full HTML pages. In Django, we use *templates* to serve individual HTML files. In the *movie* folder, create a folder called *templates* . Each app should have their own *templates* folder (fig. 6.1).

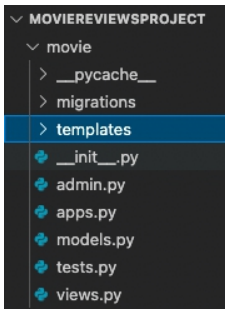


Figure 6.1

In the course of this book, you will see a pattern repeatedly in Django development: Templates, Views and URLs. We have already worked with Views and URLs in the previous chapter. The order between them doesn ' t matter, but all three are required and work closely together. Let ' s implement the template and view for *home* .

Template

In */movie/templates/* , create a new file *home.html* . This will be the full HTML page for Home. For now, fill it in with the following:

```
<h1>Welcome to Home Page</h1>
<h2>This is the full home page</h2>
```

As you can see, the template simply holds the HTML.

View

Back in */movie/views.py* , make the following change to the home function:

```
from django.shortcuts import render
```

```
from django.http import HttpResponse


def home(request):
    return render(request, 'home.html')
```

Note we are now using *render* instead of *HttpResponse* . And in *render* , we specify *home.html* instead. So we can continue to build up the HTML in *home.html* .

As you can see, the view contains the logic or the ‘ what ’ . For now, we don ’ t have much logic, but we shall explore views with more logic as we progress.

URLs

We have earlier created the URL for our *home* and *about* page in */moviereviews/urls.py* :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home),
    path('about/', movieViews.about),
]
```

The URLs control the route and entry point into a page, such as ‘ *about / ’* . You will see this pattern of Templates, Views, URLs for every Django web page. As we repeat this multiple times throughout the book, you will begin to internalize it.

Passing Data into Templates

When rendering views, we can also pass in data. Add the following in **bold** into */movie/views.py* :

```
…
def home(request):
    return render(request, 'home.html', {'name':'Greg Lim'} )
…
```

We pass in a dictionary with key-value pair {'name':'Greg Lim'} to *home.html* . And in *home.html* , we retrieve the dictionary values with:

```
<h1>Welcome to Home Page , {{ name }} </h1>
…
```

{{ name }} accesses the ‘ name ’ key in the dictionary and thus retrieve the value ‘ Greg Lim ’ . So if you run the site now and go to home, you should see:

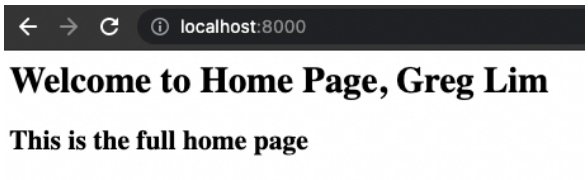


Figure 6.2

Django provide template tags e.g. {{ … }}, {%. .. %} to help render HTML. You can see the full list of built-in template tags in the official docs: (https://docs.djangoproject.com/en/3.2/ref/templates/language/).

We will introduce more template tags and their usage as we progress along.

Chapter 7: Adding Bootstrap To Our Site

Before we go any further, let ' s add bootstrap to our site. Bootstrap helps make our site look good without worrying about the necessary HTML/CSS to create a beautiful site. Bootstrap is the most popular framework for building responsive and mobile friendly websites. Instead of writing our own CSS and JavaScript, we can choose which Bootstrap component we want to use, e.g. a navigation bar, button, alert, list, card etc and simply copy and paste its markup into our template.

Go to *getbootstrap.com* and go to ' Getting Started ' (fig. 7.1).

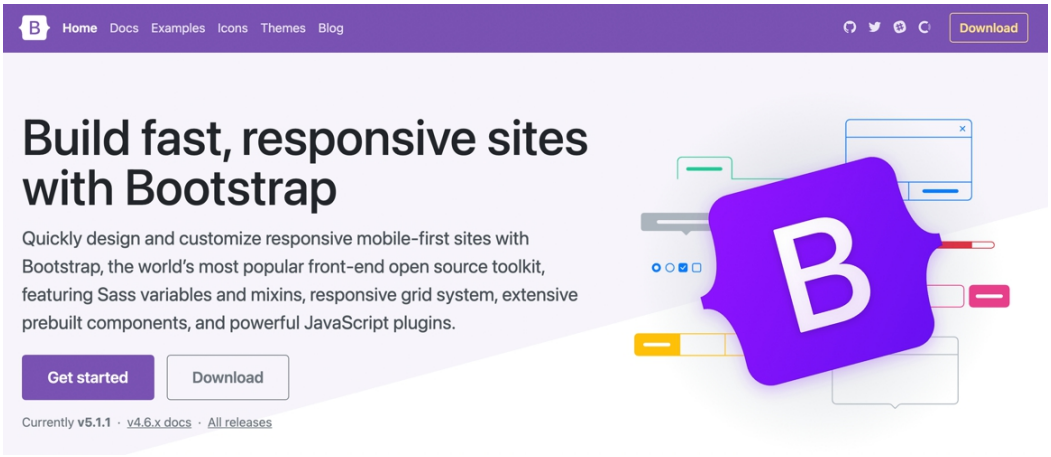


Figure 7.1

Copy the stylesheet link (fig. 7.2) into the <head> of *home.html* to load the Bootstrap CSS.

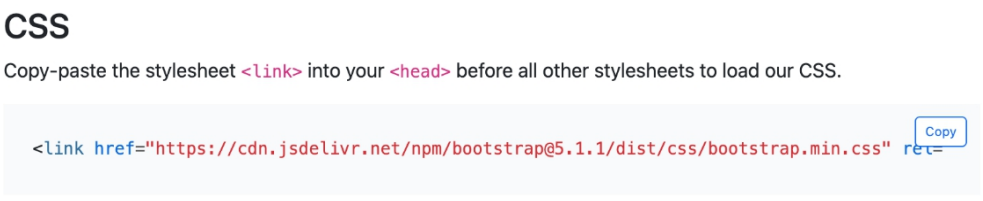


Figure 7.2

home.html will look something like:

```
<head>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css"
rel="stylesheet" crossorigin="anonymous">
</head>
<h1>Welcome to Home Page, {{ name }}</h1>
<h2>This is the full home page</h2>
```

You can immediately see the styling applied if you go to localhost:8000 (fig. 7.3):

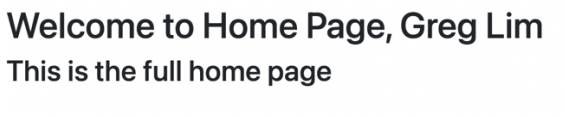


Figure 7.3

To improve the padding of the site, let ' s wrap *home.html* in a *div* container:

```
<head>
  …
</head>
<div class="container">
  <h1>Welcome to Home Page, {{ name }}</h1>
  <h2>This is the full home page</h2>
</div>
```

There are many other Bootstrap components we will use and add to our site. In the next chapter, we will use a Form from Bootstrap!

Chapter 8: Adding a Search Form

We will add a Search Form in our home page for users to search for movies. Let ' s get a Form component from *getbootstrap* . In *getbootstrap.com* , under ' Docs ' , go to ' Forms ' , ' Overview ' (fig. 8.1).

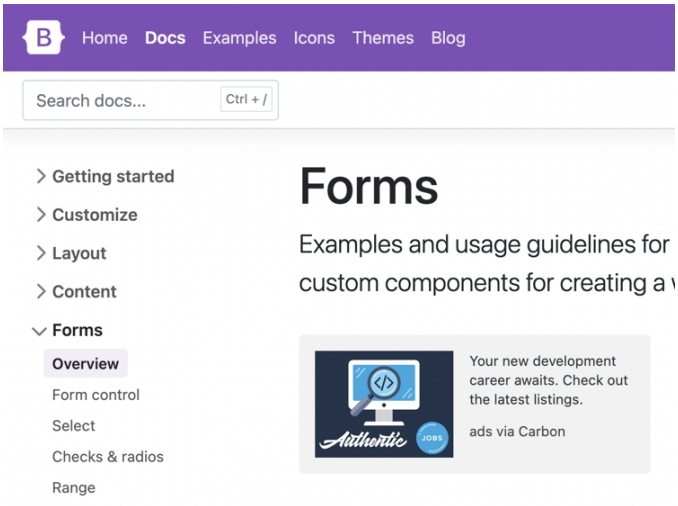


Figure 8.1

Under *Overview* , copy the markup (fig. 8.2) and paste it within the tags of *home.html* . We can remove the existing <h1> and <h2> text messages.

```
<form>
  <div class="mb-3">
    <label for="exampleInputEmail1" class="form-label">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1" aria-describedby="ema
    <div id="emailHelp" class="form-text">We'll never share your email with anyone else.</
  </div>
  <div class="mb-3">
    <label for="exampleInputPassword1" class="form-label">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1">
  </div>
  <div class="mb-3 form-check">
    <input type="checkbox" class="form-check-input" id="exampleCheck1">
    <label class="form-check-label" for="exampleCheck1">Check me out</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Figure 8.2

Because we don ' t need the password and checkbox, remove their related markup. Change the ' Email address ' to a ' Search for Movie ' field and change the ' Submit ' button to ' Search ' .

home.html will now look something like:

```
<head>
 …
</head>
<div class="container">
 <form action="">
  <div class="mb-3">
    <label for="searchMovie" class="form-label">
      Search for Movie:
    </label>
    <input type="text" class="form-control" name="searchMovie" />
  </div>
  <button type="submit" class="btn btn-primary">Search</button>
 </form>
</div>
```

And you should have a simple search form (fig. 8.3):

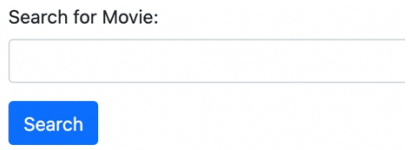


Figure 8.3

For the input, we specify *name="searchMovie"* to reference the input and retrieve its value.
We have *action=" "* in the *form* tag. The empty string *" "* specifies that upon clicking on submit, we submit the form to the same page i.e. home.html. If we want to submit the form to another page e.g. *about* , we will have: *<form action="{% url 'about' %}" >* . We will illustrate this in a later chapter. For now, we submit to the same page.

Now, how do we retrieve the values submitted? Because the form submits to *“”* , which *urls.py* routes to *def home* in *views.py* , we can retrieve the values from the *request* object in *def home* :

movie/views.py

```
…
```

```
def home(request):
    searchTerm = request.GET.get('searchMovie')
    return render(request, 'home.html', {'searchTerm':searchTerm} )
```

By default, a form submission sends a GET request if the type of request is not specified. Thus, we access the request with *request.GET* and specify the name of the input field *searchMovie* i.e. *request.GET.* ***get('searchMovie')*** to get the input value. We assign the input value to *searchTerm* .

We then pass *searchTerm* into *home.html* in the *render* function with *{'searchTerm':searchTerm}* .

And in *home.html* , we render *searchTerm* with:

```
…
<div class="container">
  <form action="">
  …
  </form>
  Searching for {{ searchTerm }}
</div>
```

When we run our app, enter a value in the Search Form and hit ‘ Search ’ . The page re-loads and the search term appears after the form (fig. 8.4).

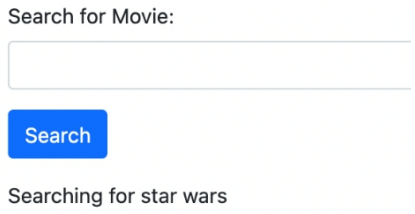


Figure 8.4

Sending a Form to Another Page

Currently, we submit a form to the same page. Suppose we want to submit a form to another page, how do we do so? Let ’ s illustrate by having a ‘ Join Our Mailing List ’ form below the Search Form. Add the below markup into *home.html* :

```
…
<div class="container">
  <form action="">
  …
  </form>
  Searching for {{ searchTerm }}
  <br />
  <br />
  <h2>Join our mailing list:</h2>
  <form action="{% url 'signup' %}">
    <div class="mb-3">
      <label for="email" class="form-label"> Enter your email: </label>
      <input type="email" class="form-control" name="email" />
    </div>
    <button type="submit" class="btn btn-primary">Sign Up</button>
  </form>
</div>
```

The signup form is similar to the search form. We have an *email* input and a *Sign Up* button. What is different is in the <form> tag, *<form action="{% url 'signup' %}" >* .

The *url* template tag *{% url 'signup' %}* takes a URL pattern name as an argument e.g. ‘ signup ’ and returns a URL link.

Let ’ s now add the ‘ signup ’ route to urlpatterns in */moviereviews/urls.py* :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home),
     path('about/', movieViews.about),
     path('signup/', movieViews.signup, name='signup'),
]
```

This time, we provide an optional URL name ‘ signup ’ to the *path* object. In doing so, we can then refer to this url name='signup ' from the url template tag *{% url 'signup' % }* . The url tag uses these names to create links for us

automatically. While its optional to add a named URL, it ' s a best practice we should adopt as it helps keep things organized as the number of URLs grows.

/movie/views.py

Next in *movie/views.py* , add the *signup* function (similar to *home*):

```
…
def signup(request):
    email = request.GET.get('email')
    return render(request, 'signup.html', {'email':email})
```

We retrieve the email from the GET request (*request.GET.get('email')*) and sends it to *signup.html* by passing in a key-value pair dictionary *{'email':email }* .

And in *movie/templates/* , create a new file *signup.html* with the below markup:

```
<h2>Added {{ email }} to mailing list</h2>
```

When you run your site, add a valid email into the signup form and click ' Sign Up ' (fig. 8.5).

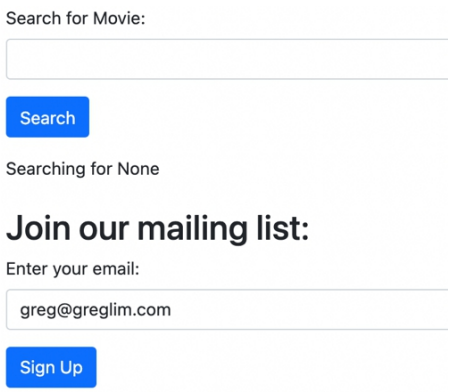


Figure 8.5

You will be brought to *signup.html* with a response message:

Added greg@greglim.com to mailing list

Note the url in signup.html. It will be something like: http://localhost:8000/signup/?email=greg%40greglim.com

This is the url that is sent in the GET request. You can see the parameters being passed in via the url. But what if you have a login form that passes in a username and password? You will want this hidden in the url. We will later see how to send a POST request from a form that hides the values passed in.

Creating a Back Link

Suppose we want to create a link from *signup.html* back to *home.html* . We can do so using the <a> tag. In *signup.html* , add the line in **bold** :

```
<h2>Added {{ email }} to mailing list</h2>
<a href="{% url 'home' %}">Home</a>
```

And in */moviereviews/urls.py* , add in **bold** :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home , name='home' ),
     path('about/', movieViews.about , name='about' ),
    path('signup/', movieViews.signup, name='signup'),
]
```

We added ***name='home'*** to configure a URL route for our homepage *{% url 'home' %}* . And we also added the ***name='about'*** to configure a URL route for our about page.

If you go to *signup.html* , there will be a link to navigate back to home (fig. 8.6).

Figure 8.6

Summary

By now, you should have a better grasp of what happens when a user types a url into the browser, sends a request to our site, goes through *urls.py* and our Django server uses a template to respond with HTML. We hope this serves as a solid foundation to move on to the next part of our project where we will go through more advanced topics like Models to make our site database-driven.

Chapter 9: Models

Working with databases in Django involves working with models. We create a database model (e.g. blog posts, movies) and Django turns these models into a database table for us.
In */movie* , you have the file *models.py* where we create our models. Fill it in with the following:

```
from django.db import models


class Movie(models.Model):
    title = models.CharField(max_length=100)
    description = models.CharField(max_length=250)
    image = models.ImageField(upload_to='movie/images/')
    url = models.URLField(blank=True)
```

Code Explanation

```
from django.db import models
```

Django imports a module *models* to help us build database models which ‘ model ’ the characteristics of the data in the database. In our case, we created a movie model to store the title, description, image and url of a movie.

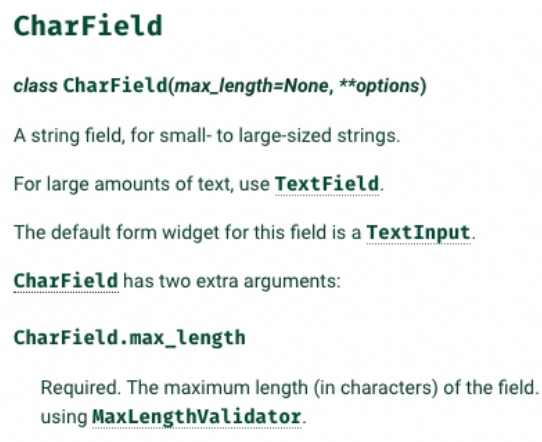
```
class Movie(models.Model)
```

class Movie inherits from the *Model* class. The *Model* class allows us to interact with the database, create a table, retrieve and make changes to data in the database.

```
    title = models.CharField(max_length=100)
    description = models.CharField(max_length=250)
    image = models.ImageField(upload_to='movie/images/')
    url = models.URLField(blank=True)
```

We then have the properties of the model. Notice that the properties have types like *CharField* , *ImageField* , *URLField* . Django provides many other model fields to support common types like dates, integers, emails and so on.

To have a complete documentation of the kinds of types and how to use them, refer to the *Model* field reference in the Django documentation (https://docs.djangoproject.com/en/3.2/ref/models/fields/). For example, CharField is a string field for small to large sized strings and the max_length arguments is required (fig. 9.1).



CharField

class **CharField**(*max_length=None*, ***options*)

A string field, for small- to large-sized strings.

For large amounts of text, use **TextField**.

The default form widget for this field is a **TextInput**.

CharField has two extra arguments:

CharField.max_length

Required. The maximum length (in characters) of the field. using **MaxLengthValidator**.

Figure 9.1

We assign *CharField* to both *title* and *description* .

```
image = models.ImageField(upload_to='movie/images/')
```

image is of *ImageField* and we specify the *upload_to* option to specify a subdirectory of MEDIA_ROOT (found in *settings.py*) to use for uploaded images.

```
url = models.URLField(blank=True)
```

url is of *URLField* , a *CharField* for a url. Because not all movies have urls, we specify *blank=True* to mean that this field is optional.

We will use this model to create a Movie table in our database.

Installing Pillow

Because we are using images, we need to install Pillow (https://pypi.org/project/Pillow/) which adds image processing capabilities to our Python interpreter.

In the Terminal, stop the server and run:

```
pip3 install pillow
```

Migrations

Currently, notice a message in the Terminal when you run the server:

“ You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them . ”

As per the message instructions, run:

```
python3 manage.py migrate
```

```
MacBook-Air:moviereviewsproject user$ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Figure 9.2

The *migrate* command creates an initial database based on Django ’ s default settings. Notice there is a *db.sqlite3* file in the project folder. The file represents our SQLite database. It is created the first time we run either *migrate* or *runserver* . *runserver* configures the database using Django ’ s default settings. But *migrate* syncs the database with the current state of any database models in the project and listed in INSTALLED_APPS. It is thus important that after we update a model, we need to run *migrate* .

Essentially, whenever we create a new model or make changes to it in *models.py* , we need to update Django in a two-step process.
First, we create a migration file with the *makemigrations* command:

```
python3 manage.py makemigrations
```

This generates the SQL commands for preinstalled apps in our INSTALLED_APPS setting. The SQL commands are not yet executed but are just a record of all changes to our models.

Second, we build the actual database with *migrate* (python3 manage.py migrat e) which executes the SQL commands in the migrations file.

The migrations are stored in an auto-generated folder *migrations* (fig. 9.3).

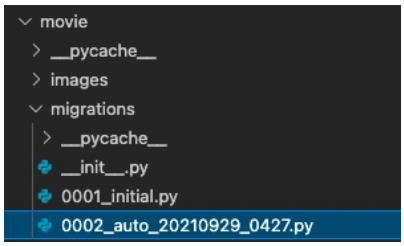


Figure 9.3

In summary, each time you make changes to a model file, you have to run:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

But how do we access our database and view what ' s inside? For that, we use a powerful tool in Django called the Admin interface. More on that in the next chapter.

* You can obtain the source code of the completed project at www.greglim.co/p/django.

Chapter 10: Django Admin Interface

To access our database, we have to go into the Django Admin interface. Remember that there is an *admin* path in */moviereviews/urls.py* ?

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home, name='home'),
    path('about/', movieViews.about, name='about'),
    path('signup/', movieViews.signup, name='signup'),
]
```

If you go to *localhost:8000/admin* , it brings you to the admin site (fig. 10.1).

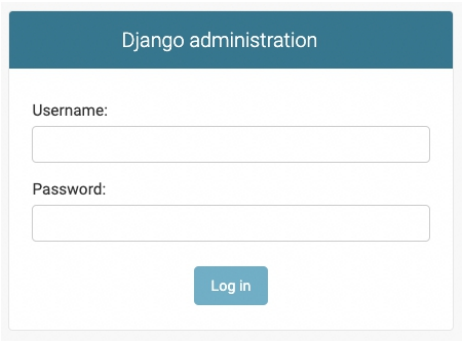


Figure 10.1

Django has a powerful built-in admin interface which provides a visual way of managing all aspects of a Django project, e.g. users, making changes to model data.

With what username and password do we log in to Admin? We will have to first create a superuser in the Terminal.

In the Terminal, run:

```
python3 manage.py createsuperuser
```

You will then be asked to specify a username, email and password. Note that

anyone can access the admin path on your site, so make sure your password is something secure.

If you wish to change your password later, you can run the command:

```
python3 manage.py changepassword <username>
```

With the username you have just created, log into admin (fig. 10.2).

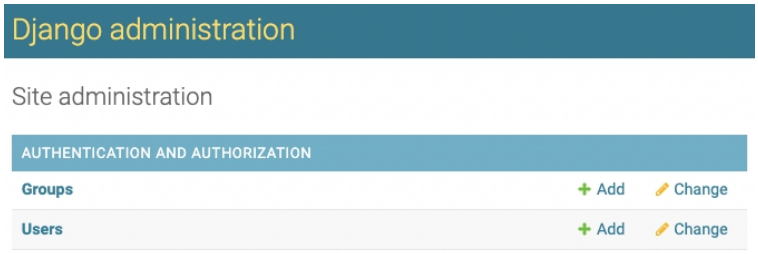


Figure 10.2

Under ‘ Users ’ , you will see the user you have just created (fig. 10.3). You can add additional user accounts here for your team.

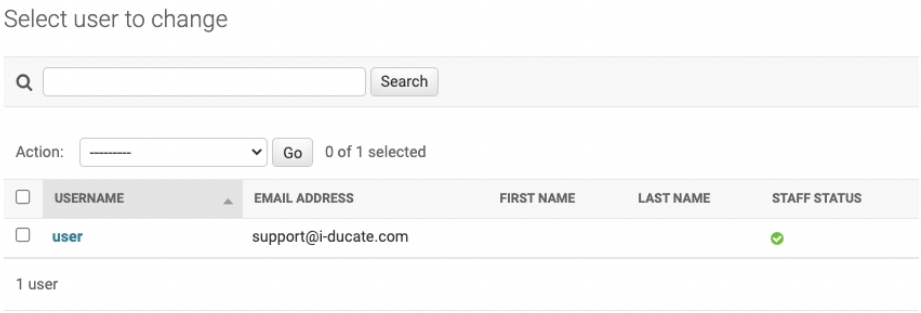


Figure 10.3

Currently, our Movie model doesn ’ t show up in admin. We need to explicitly tell Django what to display in the admin. Before adding our Movie model in admin, let ’ s first configure our images.

Configuring for Images

We have to configure where to store our images when we add them. First, go to *moviereviews/settings.py* and add at the bottom of the file:

```
…
```

```
MEDIA_ROOT = os.path.join(BASE_DIR,'media')
MEDIA_URL = '/media/'
```

And at the top of the file, add:

```
import os
```

Code Explanation

MEDIA_ROOT is the absolute filesystem path to the directory that will hold user-uploaded files. We join BASE_DIR with ‘ media ’ .
MEDIA_URL is the URL that handles the media served from MEDIA_ROOT.

(Refer to https://docs.djangoproject.com/en/3.2/ref/settings/ to find out more about the setting properties)

When we add a movie in admin (explained later), you will see the image stored inside the
/moviereviews/media/ folder.

Serving the Stored Images

Next, to enable the server serve the stored images, we have to add to */moviereviews/urls.py* :

```
…
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    …
]

urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

With this, you can serve the static media from Django.

Adding Movie model to admin

To do so, go back to */movie/admin.py* , and register our model with:

```
from django.contrib import admin
from .models import Movie


admin.site.register(Movie)
```

When you save your file, go back to Admin. The Movie model will now show up (fig. 10.4).

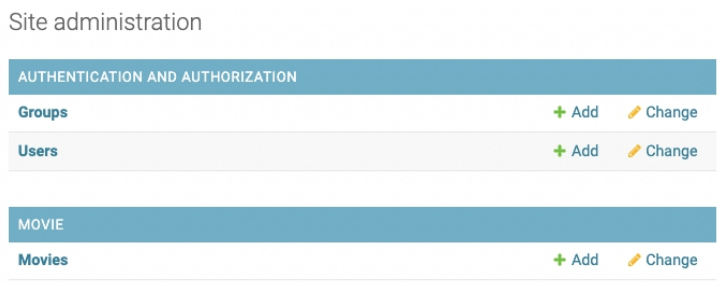


Figure 10.4

Try adding a movie object by clicking on ‘ +Add ’ . You will be brought to a Add Movie Form (fig. 10.5).

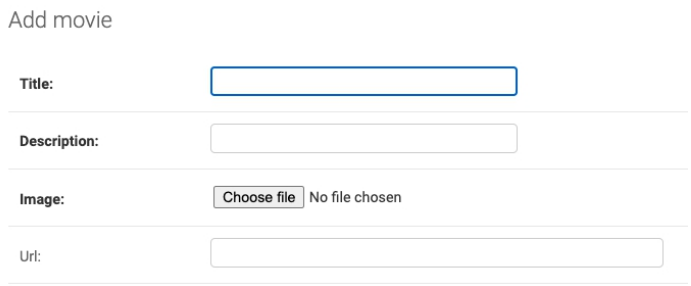


Figure 10.5

Note that *Url* is not in bold as we have marked it as optional back in *models.py* , *url = models.URLField(blank=True)* . The other required fields are in **bold** .

Try adding a movie and hit ‘ Save ’ .

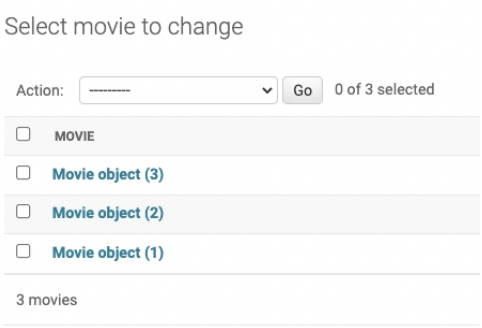


Figure 10.6

Your movie object will be saved to the database and reflected in the admin (fig. 10.6), and you can see the movie image in:
/moviereviews/media/movie/images/<image file>.jpg
Now that we know how to add movie objects to our database via admin, let ' s see how we can display them in our site in the next chapter.

Chapter 11: Displaying Objects From Admin

In this chapter, we will see how to display the movie objects stored in the admin database. In *movie/views.py* , add the following in **bold** :

```
…
from .models import Movie

def home(request):
    searchTerm = request.GET.get('searchMovie')
    movies = Movie.objects.all()
   return render(request, 'home.html',          {'searchTerm':searchTerm , 'movies': movies })
…
```

Code Explanation

```
from .models import Movie
```

We first import the *Movie* model.

```
movies = Movie.objects.all()
```

The above grabs all the movie objects from the database and assigns it to *movies* . We then pass movies in the dictionary to *home.html* .

Django makes it really straightforward to access objects in the database. If we have to write code to connect to the database, write SQL statements for retrieval, and convert the results to Python objects, it would involve a lot more code! But Django provides lots of database functionality to handle these for us.

/movie/templates/home.html

And in *home.html* , we display the objects by adding the following code in **bold** :

```
…
```

```
<div class="container">
 <form action="">
 …
 </form>
 <p> Searching for: {{ searchTerm }} </p>
 {% for movie in movies %}
  <h2>{{ movie.title }}</h2>
  <h3>{{ movie.description }}</h3>
  <img src="{{ movie.image.url }}">
  {% if movie.url %}
   <a href="{{ movie.url }}">Movie Link</a>
  {% endif %}
 {% endfor %}
 <br />
 <br />
  <h2>Join our mailing list:</h2>
 …
```

Code Explanation

```
{% for movie in movies %}
 …
{% endfor %}
```

Using a *for* -loop in the Django templating language, we loop through *movies* with *movie* acting as a temporary variable to hold the element for the current iteration. Note that we enclose code in {% … %} tags. We use {{ … }} to render variables like a movie ' s title, description and image url. For example:

```
<h2> {{ movie.title }} </h2>
<h3> {{ movie.description }} </h3>
<img src=" {{ movie.image.url }} ">
```

Because a movie url is optional, i.e. it can be null, we check if it has a value with {% if movie.url % } , and if so, render a <a> href to the movie url.

```
{% if movie.url %}
  <a href="{{ movie.url }}">Movie Link</a>
```

```
{% endif %}
```

Running your App

When you run your site, and go to the home page, you will see the movies you added (in Admin) on the page (fig. 11.1).

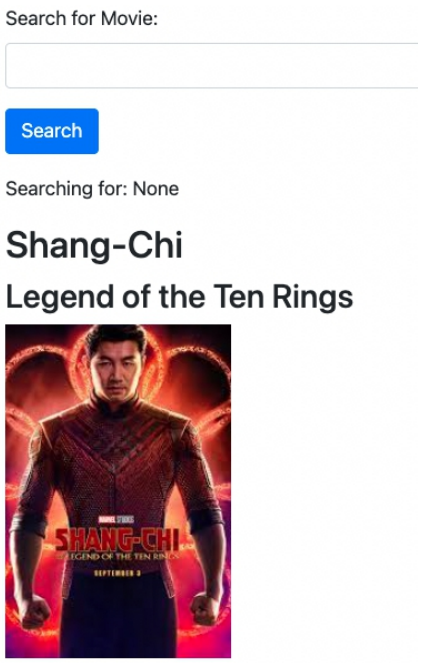


Figure 11.1

If you add another movie in the Admin, it will be listed in the website when you re-load it.

Using the Card Component

Let ' s further improve the look of our site by using the Card component from Bootstrap to display each movie (https://getbootstrap.com/docs/5.1/components/card/).

Each movie will be displayed in a Card component. In *movie/templates/home.html* , replace the *for* loop markup and make the following changes in **bold** :

```
…
<div class="container">
 <form action="">
```

```
…
</form>
<p>Searching for: {{ searchTerm }}</p>
<div class="row row-cols-1 row-cols-md-3 g-4">
  {% for movie in movies %}
  <div v-for="movie in movies" class="col">
    <div class="card">
      <img class="card-img-top" src="{{ movie.image.url }}" />
      <div class="card-body">
        <h5 class="card-title fw-bold">{{ movie.title }}</h5>
        <p class="card-text">{{ movie.description }}</p>
        {% if movie.url %}
        <a href="{{ movie.url }}" class="btn btn-primary">
          Movie Link
        </a>
        {% endif %}
      </div>
    </div>
  </div>
  {% endfor %}
</div>
<br />
<br />
 <h2>Join our mailing list:</h2>
…
```

Your site should look something like (fig. 11.2):

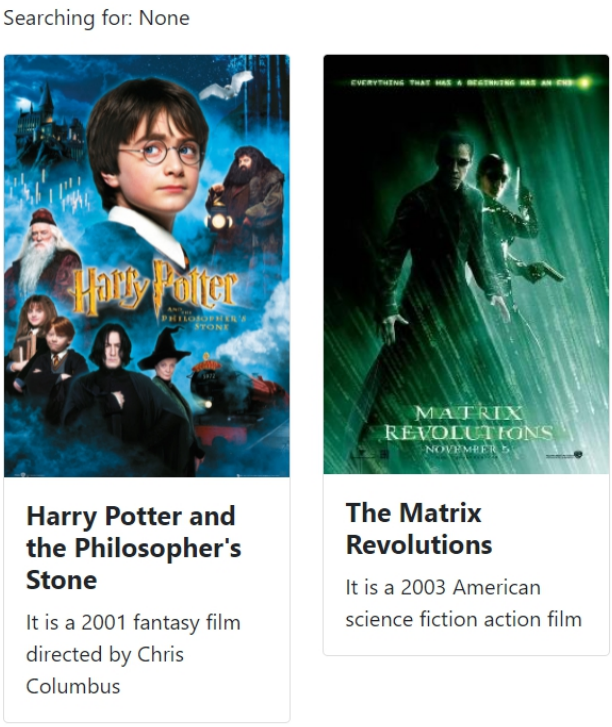


Figure 11.2

Implementing Search

We are currently listing all movies in our database. Let ' s display only the movies that fits the user-entered search term. Implement *def home* in */movie/views.py* with the following:

```
def home(request):
    searchTerm = request.GET.get('searchMovie')
    if searchTerm:
        movies = Movie.objects.filter(title__icontains=searchTerm)
    else:
        movies = Movie.objects.all()
    return render(request, 'home.html',          {'searchTerm':searchTerm, 'movies': movies})
```

Code Explanation

```
    searchTerm = request.GET.get('searchMovie')
```

We retrieve the search term entered (if any) from the *searchMovie* input.

```
    if searchTerm:
        movies = Movie.objects.filter(title__icontains=searchTerm)
```

If a search term is entered, we call the model ' s *filter* method to return the movie objects with a case-insensitive match to the search term.

```
else:
    movies = Movie.objects.all()
```

If there is no search term entered, we simply return all movies.

Running your App

When you run your app and enter a search term, the site displays only the movies that fits the search term.

Chapter 12: Revisiting Concepts - Adding A News App

Up till this point, we have covered a lot of material. Let ' s now crystalize and recap on the concepts learned by adding a News app to our site. We currently have one app, Movie in our project. Let ' s add a News app. Do you remember how to add an app, add a model, and then display the objects from the Admin database? Try it on your own as a challenge.

Have you tried it? Let ' s now go through it together. To add a News app, run in the Terminal:

```
python3 manage.py startapp news
```

A *news* folder will be added to the project. Each time we add an app, we have to tell Django about it by adding it to *settings.py* :

/moviereviews/settings.py

```
…
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'movie',
    'news',
]
…
```

Next, we have to add the path to *news* in */moviereviews/urls.py* . Notice that in *urls.py* , we have quite a few existing paths for the movie app.

```
…
urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('', movieViews.home, name='home'),
    path('about/', movieViews.about, name='about'),
    path('signup/', movieViews.signup, name='signup'),
]
…
```

If we were to add the paths for news, the number of paths will increase and it will soon be difficult to distinguish which paths are for which app (especially when the project grows). To better segregate the paths into their own apps, each app can have their own *urls.py* .

First, add in */moviereviews/urls.py* :

```
…
from django.contrib import admin
from django.urls import path , include
…

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home, name='home'),
    path('about/', movieViews.about, name='about'),
    path('signup/', movieViews.signup, name='signup'),
     path('news/', include('news.urls')),
]
…
```

path('news/', include('news.url')) will forward any requests with ‘ news/ ’ to *news* app ’ s *urls.py* .

In */news* , create a new file *urls.py* with the following:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.news, name='news'),
]
```

The above path forwards a request, e.g. *localhost:8000/news* to the news view.

Next in */news/views.py* , add the *def news* function:

```
from django.shortcuts import render

def news(request):
    return render(request, 'news.html')
```

In */news* , we now need to create the *templates* folder, and in it, a new file *news.html* . We will later populate this file to display news articles from the admin database.

Let ' s first create the News model.

News Model

In */news/models.py* , create the model with the following:

```
from django.shortcuts import render

class News(models.Model):
    headline = models.CharField(max_length=200)
    body = models.TextField()
    date = models.DateField()
```

Because we have added a new model, we need to make migrations with:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Next, register the news model by going to */news/admin.py* and adding:

```
from django.contrib import admin
from .models import News

admin.site.register(News)
```

When you run the server and go to Admin now, it should reflect the news model.

Displaying the news articles

Let ’ s go ahead and display the news articles in *news.html* .

In */news/views.py* , add in the codes in **bold** :

```
from django.shortcuts import render
from .models import News

def news(request):
    newss = News.objects.all()
    return render(request, 'news.html' , {'newss':newss} )
```

We retrieve the news objects from the database and then pass it to *news.html* .

In */news/templates/news.html* , display the news objects with:

```
{% for news in newss %}
<h2>{{ news.headline }}</h2>
<h5>{{ news.date }}</h5>
<p>{{ news.body }}</p>
{% endfor %}
```

Running Our App

Now, try adding some news objects in admin. And when you visit http://localhost:8000/news/ , you should see them displayed (fig. 12.1).

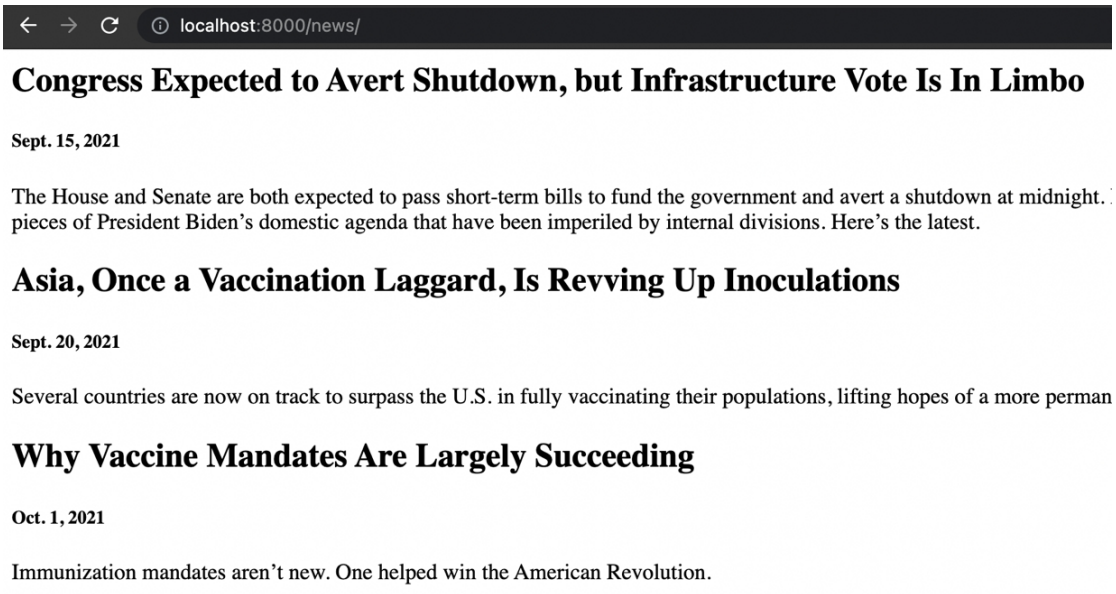


Figure 12.1

When displaying news, we should be displaying the most recent news first. To do this, we can order the news objects in */news/views.py* by specifying *order_by* :

```
…
def news(request):
    newss = News.objects.all(). order_by('-date')
   return render(request, 'news.html' , {'newss':newss})
```

The most recent news will now be displayed first (fig. 12.2):

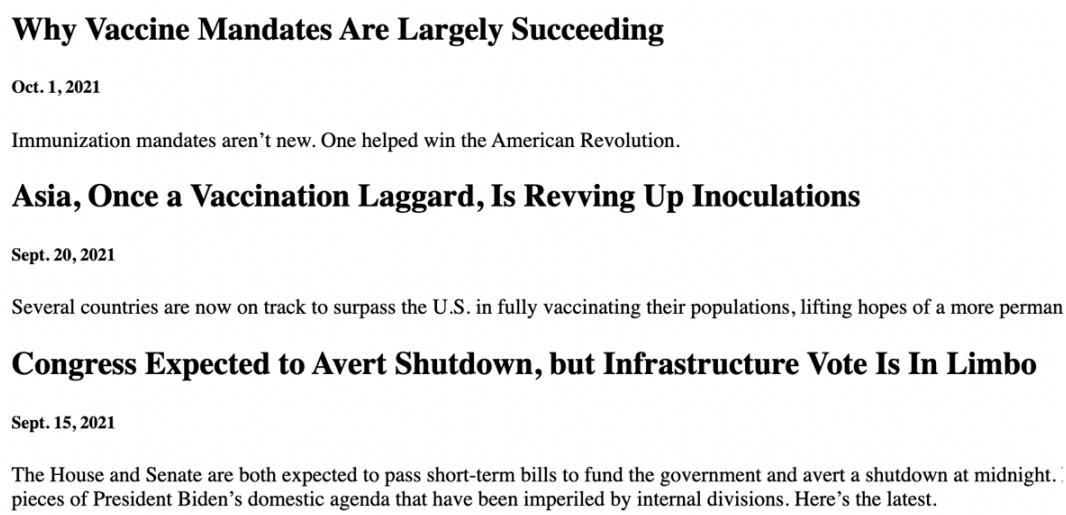


Figure 12.2

We should also improve the look of the news site with the Bootstrap Horizontal Card component. Note that you need to include the bootstrap CSS link in your <head>. Replace the entire */news/templates/news.html* code with the following code:

```
<head>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css"
rel="stylesheet" crossorigin="anonymous">
</head>
<div class="container">
  {% for news in newss %}
  <div class="card mb-3">
    <div class="row g-0">
      <div>
        <div class="card-body">
          <h5 class="card-title"> {{ news.headline }} </h5>
          <p class="card-text"> {{ news.body }} </p>
          <p class="card-text"><small class="text-muted">
             {{ news.date }}
           </small></p>
        </div>
      </div>
    </div>
  </div>
  {% endfor %}
</div>
```

And you should get something like (fig. 12.3):

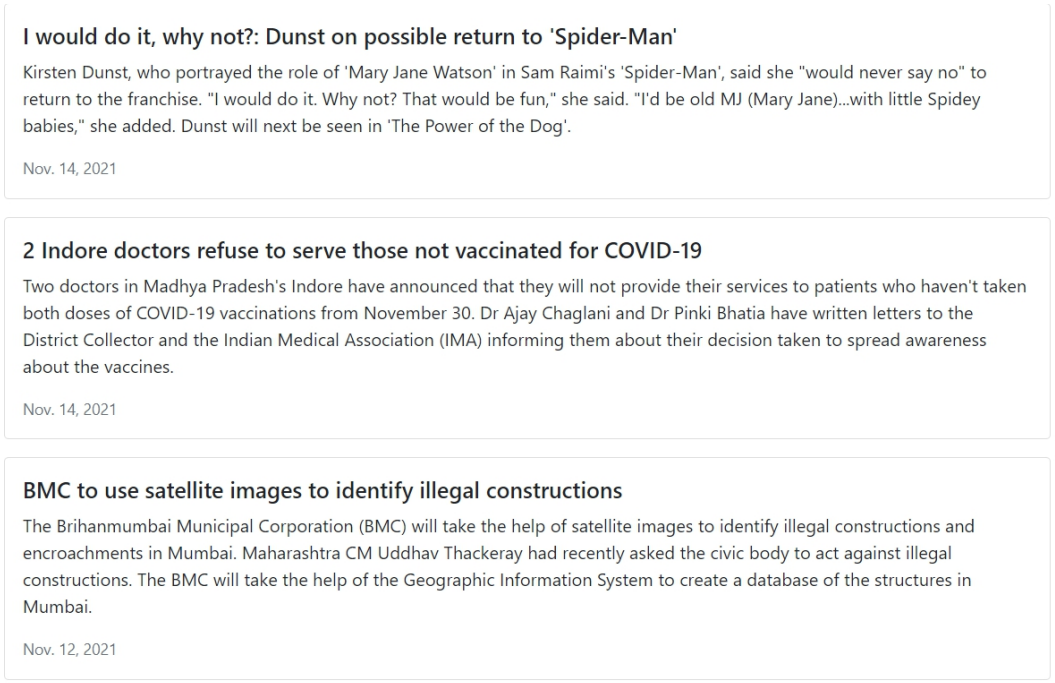


Figure 12.3

We hope that this chapter crystalizes your understanding of adding an app to the project, adding a model and displaying the model objects from the database in the template. In the next chapter, we go deeper in understanding how the database works.

Chapter 13: Understanding the Database

Let ' s take some time to understand how the database works. The objects are stored in the *db.sqlite3* file. If you click on it, it is not very readable. But you can view such *sqlite* files with a SQLite Viewer. Just Google ' SQLite Viewer ' for a list of them. One example is https://inloop.github.io/sqlite-viewer/ .

Open your *sqlite3* file with the viewer and you can see the different tables in the database (fig. 13.1).

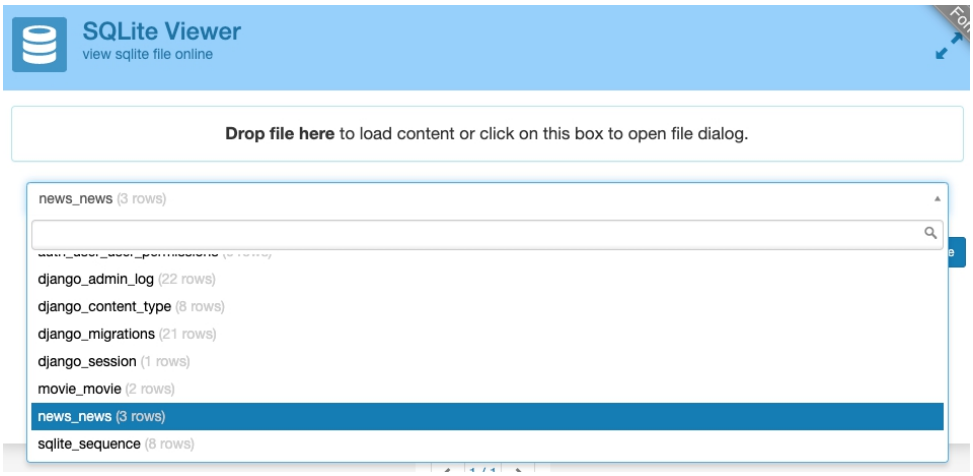


Figure 13.1

You can see the tables of the models we have created i.e. *movie* and *news* . There are other tables like *django_session* because of the different apps that are installed for functions like sessions and authentications.

Select a table e.g. *news_news* and you can see its rows (fig. 13.2).
Note: the table name is derived from <appname>_<modelname>. We can have multiple models in a app. Eg *movie_movie* , *movie_review*

Figure 13.2

Note that each row has a Django generated unique id as primary key. Django automatically adds a primary key to each table.

So hopefully this lets you appreciate what goes on behind the scenes in a Django database. Currently, we are using an SQL based database. What if we want to switch to some other database e.g. NoSQL, PostgreSQL, Oracle, MySQL? Django provides built-in support for several types of database backends.

You should go to */moviereviews/settings.py* , and make changes to the lines in **bold** (to switch to another database engine):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

You can still create your models as per normal and the changes are handled by Django behind the scenes.

In the book, we are using SQLite because it is the simplest. Django uses SQLite by default and it ' s a great choice for small projects. It runs off a single file and doesn ' t require complex installation. In contrast, the other options involve some complexity to configure them properly. In the last chapter, we go through the process of switching to MySQL as our production database.

Chapter 14: Displaying Objects in Admin

Currently when we look at our model objects from Admin, it is hard to identify the individual objects (fig. 14.1) e.g. News object (1), News object (2)

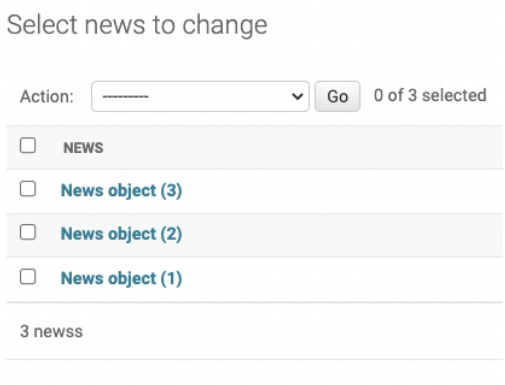


Figure 14.1

For better readability in the Admin, we can customize what is displayed there. For e.g. if we want to display the headline for each news object instead, in */news/models.py* , add the function in **bold** :

```
from django.db import models


class News(models.Model):
    headline = models.CharField(max_length=200)
    body = models.TextField()
    date = models.DateField()

    def __str__(self):
        return self.headline
```

The *__str__* method in Python represents the class objects as a string. *__str__* will be called when the news objects are listed in Admin. See how readability is improved (fig. 14.2)!

Figure 14.2

Note that we don ’ t need to do any migration since no data is changed. We have just added a function which returns data.

Chapter 15: Extending Base Templates

We currently have our movies page, mailing list signup page and news page. But users have to manually enter in the url to navigate to each of the pages which is not ideal. Let ' s add a header bar that allows them to navigate between pages. We begin with *movie/templates/home.html* .

We use as a base the markup of the Navbar component from getbootstrap (fig. 15.1).

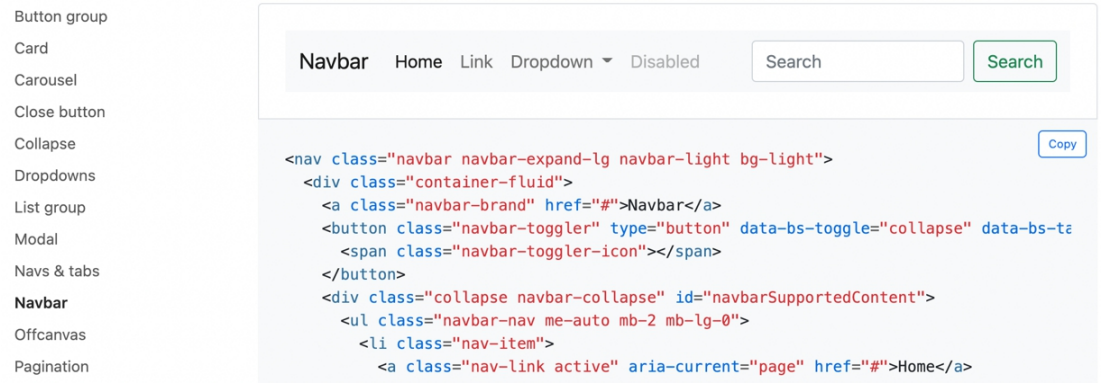


Figure 15.1

We also include the *bootstrap.bundle.min.js* script inside the <head> tag. This file provides additional user interface elements such as dialog boxes, tooltips, carousels, and button interactions.

In *movie/templates/home.html* , make the following changes in **bold** :

```
…
 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js"
crossorigin="anonymous">
 </script>
</head>

<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
 <div class="container">
  <a class="navbar-brand" href="#">Movies</a>
```

```
  <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navbarNavAltMarkup"
    aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle
navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
    <div class="navbar-nav ms-auto">
      <a class="nav-link" href="#">News</a>
      <a class="nav-link" href="#">Login</a>
      <a class="nav-link" href="#">Sign Up</a>
    </div>
  </div>
 </div>
</nav>

…
```

We have added the navbar to *home.html* (fig. 15.2). We also included two links ("Login", and "Sign Up") which will be used later.

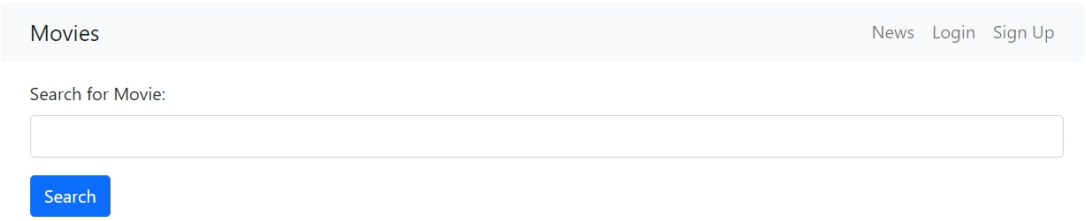


Figure 15.2

If you reduce the browser window width, your navbar will respond accordingly (this is automatically provided by the Bootstrap elements we used – fig. 15.3).

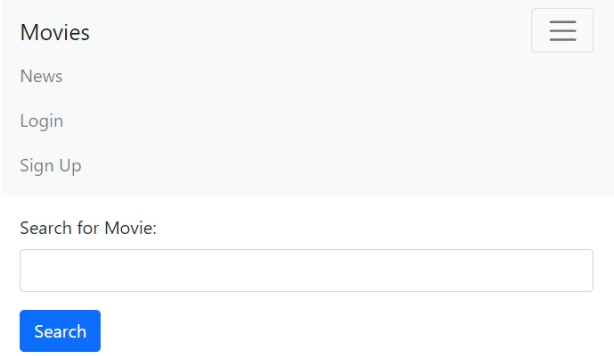


Figure 15.3

But should we repeat the same process and copy the exact same code into *news.html* and other future pages?

This would duplicate a lot of the same code. Worse, it will be very hard to maintain the code. Suppose we want to add a new link, we have to add the link in multiple pages!

To fix this, we will be using base templates where we can add the navbar to every single page. This allows us to make changes to our navbar in a single place and it will apply to every page.

Since this will be a “global" template (which will be used across all pages and apps), we will add it to the main folder (*moviereviews*). In the *moviereviews* folder, create a folder called *templates* . In that folder, create a file *base.html* . Move the entire <header> and <nav> markups from *home.html* into *base.html* . (Note that we can name *base.html* anything, but using *base.html* is a common convention for base templates).

Then, add the following at the end of the file:

```
<div class="container">
  {% block content %}
   {% endblock content %}
</div>
```

The entire *moviereviews/templates/base.html* will look something like:

```
<head>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet"
crossorigin="anonymous" />
```

```
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js"
crossorigin="anonymous">
  </script>
</head>

<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
  <div class="container">
    <a class="navbar-brand" href="#">Movies</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navbarNavAltMarkup"
      aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
      <div class="navbar-nav ms-auto">
        <a class="nav-link" href="#">News</a>
        <a class="nav-link" href="#">Login</a>
        <a class="nav-link" href="#">Sign Up</a>
      </div>
    </div>
  </div>
</nav>

<div class="container">
  {% block content %}
  {% endblock content %}
</div>
```

Finally, we need to register *moviereviews/templates* folder in our application settings. Make sure to add it to TEMPLATES DIRS in */moviereviews/settings.py* :

```
…
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [ os.path.join(BASE_DIR, 'moviereviews/templates') ],
…
```

Code Explanation

base.html as its name suggests serves as the base for all pages. Thus, we include the header navbar in it. If your site has a footer, you can also include it in.

```
{% block content %}
{% endblock content %}
```

We then allocate a 'block' where content can be slotted in from other child pages e.g. *home.html* , *news.html* . This will become clear as we proceed on.

home.html

In *home.html* , we shouldn't have the header and navbar anymore. To simplify things, let's remove the sign-up mailing list form. Also remove the tag since it is loaded in the *base.html* template. The entire *movie/templates/home.html* will look something like:

```
{% extends 'base.html' %}
{% block content %}
<form action="">
  <div class="mb-3">
    <label for="searchMovie" class="form-label"> Search for Movie: </label>
    <input type="text" class="form-control" name="searchMovie" />
  </div>
  <button type="submit" class="btn btn-primary">Search</button>
</form>
<p>Searching for: {{ searchTerm }}</p>
<div class="row row-cols-1 row-cols-md-3 g-4 mb-3">
  {% for movie in movies %}
  <div v-for="movie in movies" class="col">
    <div class="card">
      <img class="card-img-top" src="{{ movie.image.url }}" />
      <div class="card-body">
        <h5 class="card-title fw-bold">{{ movie.title }}</h5>
        <p class="card-text">{{ movie.description }}</p>
        {% if movie.url %}
```

```
      <a href="{{ movie.url }}" class="btn btn-primary">Movie Link</a>
      {% endif %}
    </div>
  </div>
 </div>
 {% endfor %}
</div>
{% endblock content %}
```

Code Explanation

```
{% extends 'base.html' %}
```

With the *extends* method, we extend from *base.html* by taking all the markup inside the *block content* tag in *home.html* and put it into *base.html* .

Running your app

When you run the app and go to the home page, you should see the same site with the navbar included magically as before!

/news/template/news.html

Now, let's apply the above to *news.html* as well. In */news/template/news.html* , remove the <head> element, remove the tag, and add the codes in **bold** :

```
{% extends 'base.html' %}
{% block content %}
{% for news in newss %}
<div class="card mb-3">
 <div class="row g-0">
  <div>
   <div class="card-body">
    <h5 class="card-title">{{ news.headline }}</h5>
    <p class="card-text">{{ news.body }}</p>
    <p class="card-text">
     <small class="text-muted">{{ news.date }}</small>
    </p>
   </div>
```

```
    </div>
  </div>
</div>
{% endfor %}
{% endblock content %}
```

When you run your app, the news page should show the navbar as well. And now, whenever you want to make a change to navbar, you just have to do it once in *base.html* and it will apply for all the pages!

Making the Links Work

The links in the navbar don't currently work. To do so, in *base.html* , add the codes in **bold** :

```
…
<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
  <div class="container">
    <a class="navbar-brand" href=" {% url 'home' %} ">Movies</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navbarNavAltMarkup"
      aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
      <div class="navbar-nav ms-auto">
        <a class="nav-link" href=" {% url 'news' %} ">News</a>
        <a class="nav-link" href="#">Login</a>
        <a class="nav-link" href="#">Sign Up</a>
      </div>
    </div>
  </div>
</nav>
…
```

When you run your app, the links will work. This is because we have earlier defined the 'home' path in */moviereviews/urls.py* :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
     path('', movieViews.home, name='home'),
]
…
```

And the ‘news’ path in */news/urls.py* :

```
…
urlpatterns = [
    …
     path('news/', include('news.urls')),
]
…
```

When you run your app now, you can navigate between the Movie and News page using the links in the navbar.

Adding a Footer Section

Let’s add a footer section inside our *base.html* . In *moviereviews/templates/base.html* , we make the following changes in **bold** :

```
…
<div class="container">
  {% block content %}
   {% endblock content %}
</div>

<footer class="text-center text-lg-start bg-light text-muted mt-4">
  <div class="text-center p-4">
    © 2021 Copyright -
    <a class="text-reset fw-bold text-decoration-none"
      target="_blank" href="https://twitter.com/greglim81">
      Greg Lim
    </a> -
    <a class="text-reset fw-bold text-decoration-none"
      target="_blank" href="https://twitter.com/danielgarax">
```

```
    Daniel Correa
  </a>
 </div>
</footer>
```

The footer section displays a grey div in where we place the book authors' names, with a link to their respective Twitter account links. If you run the app now, it should give you something like this (fig. 15.4):

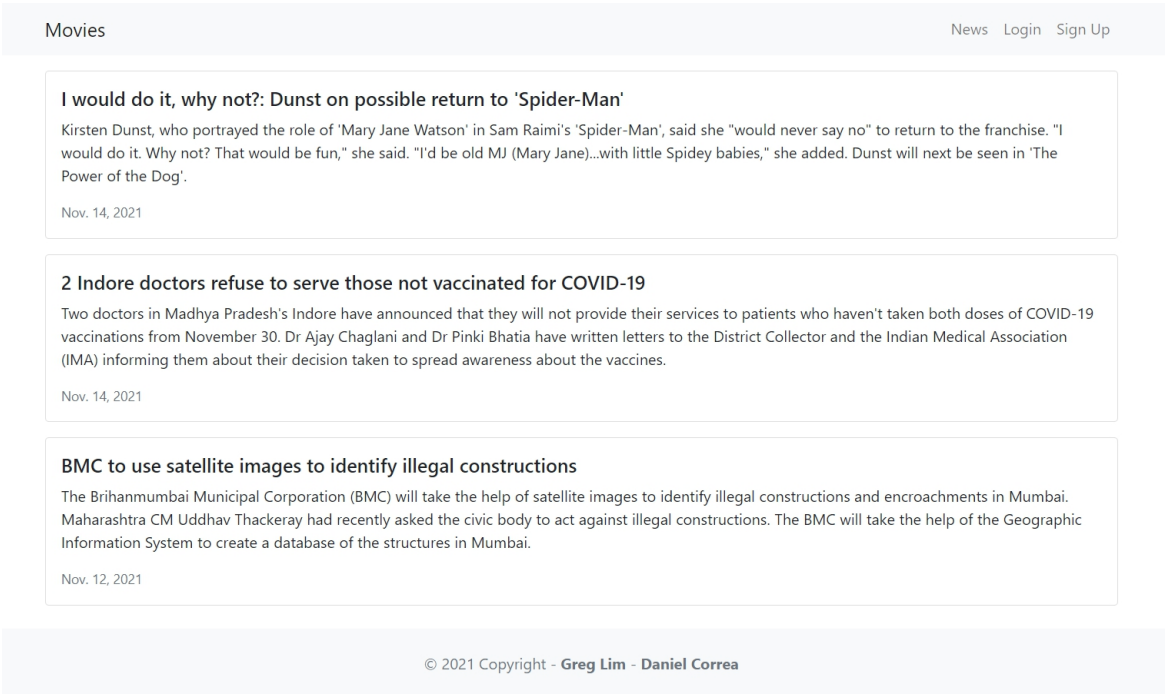


Figure 15.4

In the next chapter, we look at how to display static images on our site.

Chapter 16: Static Files

Suppose we want to display an image icon for our site (fig. 16.1):

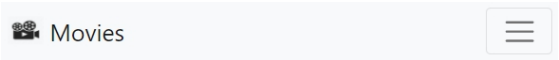


Figure 16.1

Such are examples of fixed images on the site. These fixed images are static files. They are different from media files which users upload to the site e.g. movie images.

In */moviereviews/settings.py* , we have a property STATIC_URL = ' /static/ ' . Above the property is a comment containing a link to the documentation on how to use static files:

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC_URL = '/static/'
```

But we will go through it here. In */moviereviews* , create a folder *static* . In it, create a folder *images* to contain the fixed images used in our site. Bring in an image file (e.g. *movie.png*) that you want to display in your site into the *images* folder. And in *base.html* , replace the Movies link with the following:

```
…
<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
 <div class="container">
   <a class="navbar-brand" href="{% url 'home' %}">
    {% load static %}
    <img src="{% static 'images/movie.png' %}" alt="" width="30" height="24"
     class="d-inline-block align-text-top" />
    Movies
   </a>
…
```

Code Explanation

We added an image to the navbar (https://getbootstrap.com/docs/5.1/components/navbar/).

```
    {% load static %}
```

To add static files to our template, we add the above line to the top of *base.html* . Because other templates inherit from *base.html* , we only have to add this once.

```
<img src="{% static 'images/movie.png' %}" …
```

In the image source, we can then reference the static image.

Since, the *movie.png* image is not attached to a specific app (it is attached to the project folder), we need to include the *moviereviews/static* folder to the application settings.

Add at the end of the */moviereviews/settings.py* file, the following code in **bold** :

```
…

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
    'moviereviews/static/',
]
```

Now, the brand image will then be displayed in the navbar (fig. 16.2).

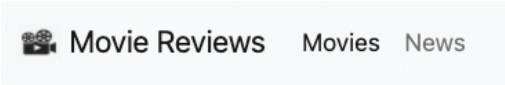


Figure 16.2

Other than images, you can link to other static file types e.g. pdf, CSS, JavaScript files. Let’s now go back to our movies app. In the next chapter, let’s see how to navigate to a movie’s detail page from the home page.

Chapter 17: Movie Detail Page

Let's now go back to our movies app. In this chapter, we want to navigate to a movie's detail page when a user clicks on a movie title on the home page. First, we add the movie details path in */moviereviews/urls.py* :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home, name='home'),
    path('about/', movieViews.about, name='about'),
    path('signup/', movieViews.signup, name='signup'),
    path('news/', include('news.urls')),
     path('movie/', include('movie.urls')),
]
…
```

path('movie/', include('movie.urls')) will forward any requests with ' movie/ ' to movie app ' s *urls.py* . We put all the paths related to the movie app in its own *urls.py* , i.e. */movie/urls.py* .

So let ' s create */movie/urls.py* with the following code:

```
from django.urls import path
from . import views

urlpatterns = [
    path('<int:movie_id>', views.detail, name='detail'),
]
```

The path above is the primary key for the movie represented as an integer *<int:movie_id >* . Remember that Django adds an auto-incrementing primary key to our database models under the hood.

With this path, when a user comes to a url e.g. *localhost:8000/movie/4* , ' 4 ' is the integer (int) representing the movie id. The url matches *path('movie/<int:movie_id>'* and navigates to the detail page.

Next in */movie/views.py* , we add the view *def detail* :

```
from django.shortcuts import render
from django.http import HttpResponse
from django.shortcuts import get_object_or_404
…

def detail(request, movie_id):
    movie = get_object_or_404(Movie,pk=movie_id)
    return render(request, 'detail.html', {'movie':movie})
```

We use get_object_or_404 to get the specific movie object we want. We provide *movie_id* as the primary key pk=movie_i d . If there is a match, *get_object_or_404* as its name suggests, returns us the object or the *not found (404)* object.

We then pass the movie object to *detail.html* . So in */movie/templates/* , create a file *detail.html* . We will use the same Horizontal Card layout from *news/templates/news.html* and extend from *base.html* . The code in */movie/templates/detail.html* will look like:

```
{% extends 'base.html' %}
{% block content %}
<div class="card mb-3">
  <div class="row g-0">
    <div class="col-md-4">
      <img src="{{ movie.image.url }}" class="img-fluid rounded-start" alt="">
    </div>
    <div class="col-md-8">
      <div class="card-body">
        <h5 class="card-title">{{ movie.title }}</h5>
        <p class="card-text">{{ movie.description }}</p>
        <p class="card-text">
          {% if movie.url %}
            <a href="{{ movie.url }}" class="btn btn-primary">
              Movie Link
            </a>
          {% endif %}
        </p>
      </div>
```

```
  </div>
 </div>
</div>
{% endblock content %}
```

We hope you are noticing the repeating pattern of creating a new view, url and template.

If you visit a movie's detail url e.g. *localhost:8000/movie/4* , it will render the movie's details in its own page (fig. 17.1).

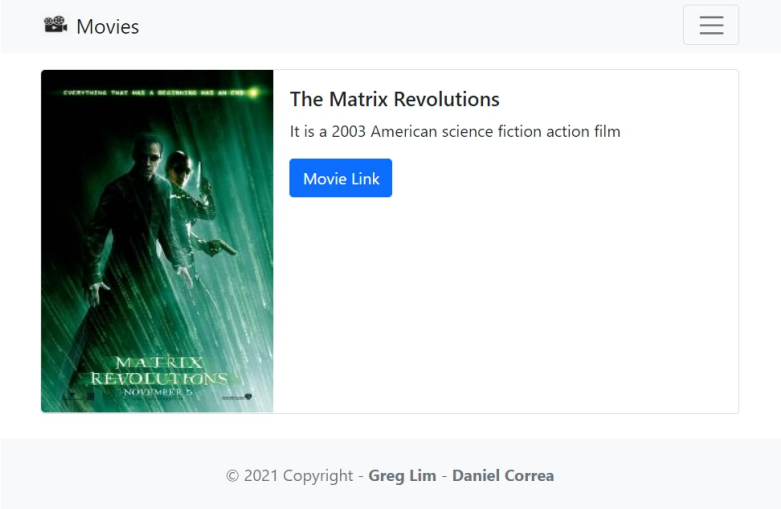


Figure 17.1

We will later add a Reviews section to the details page.

Implement Links to Individual Movie Detail Page

Now that we have a movie ' s detail page, we next implement the movie links to navigate to those detail pages from *home.html* . We simply wrap the movie title in a <a href … > as shown below:

```
…
<div class="row row-cols-1 row-cols-md-3 g-4 mb-3">
 {% for movie in movies %}
 <div v-for="movie in movies" class="col">
  <div class="card">
   <img class="card-img-top" src="{{ movie.image.url }}" />
```

```
<div class="card-body">
  <a href="{% url 'detail' movie.id %}">
   <h5 class="card-title fw-bold">{{ movie.title }}</h5>
  </a>
 <p class="card-text">{{ movie.description }}</p>
 …
```

{% url 'detail' movie.id % } links to the *detail* path back in */movie/urls.py* . Within the {% … %} tag, we have specified the target name of our url ' detail ' , and also passed *movie.id* as a parameter. When you run your app now and go to home, the title will appear as a link to the detail page (fig. 17.2).

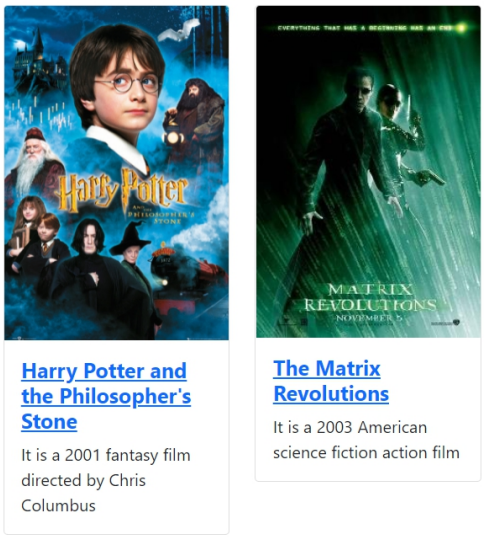


Figure 17.2

We have learned a lot up till now! We learned about databases, models, admin interface, static files, media files, extending base templates, urls, routing urls and many more.

In the next chapters, we will learn how to allow a user to sign up, log in, and

then post reviews for movies.

Chapter 18: Creating a Signup Form

The next part of our app will concern user authentication where we allow users to log in and post reviews about movies. Implementing user authentication is famously hard. Fortunately, we can use Django's powerful, built-in authentication system that takes care of the many security gotchas that can arise if we were to create our own user authentication from scratch.

In our website, if users do not yet have an account, they will have to sign up for one first. So, let's look at how to create a signup account form.

Since sign up account doesn't belong to the movie or news app, let's create a dedicated app called *accounts* for it. In the Terminal, run:

```
python3 manage.py startapp accounts
```

Make sure to add the new app to INSTALLED_APPS in */moviereviews/settings.py* :

```
…
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'movie',
    'news',
     'accounts',
]
…
```

We create a project-level url for the *accounts* path in */moviereviews/urls.py* :

```
…
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', movieViews.home, name='home'),
    path('about/', movieViews.about, name='about'),
    path('signup/', movieViews.signup, name='signup'),
    path('news/', include('news.urls')),
    path('movie/', include('movie.urls')),
     path('accounts/', include('accounts.urls')),
]
…
```

We put all the paths related to the *accounts* app (e.g. signup, login, logout) in its own *urls.py* , i.e. */accounts/urls.py* (create this file).

```
from django.urls import path
from . import views

urlpatterns = [
    path('signupaccount/', views.signupaccount, name='signupaccount'),
]
```

Next, create *def signupaccount* in */accounts/views.py* with the below codes:

```
from django.shortcuts import render
from django.contrib.auth.forms import UserCreationForm


def signupaccount(request):
    return render(request, 'signupaccount.html',
                  {'form':UserCreationForm})
```

We import *UserCreationForm* which Django provides to easily create a sign up form to register new users. Django Forms (is a vast topic by itself) and we will see how powerful they can be. We pass in the form to *signupaccount.html* .

Next, create */accounts/templates/signupaccount.html* and simply fill in:

```
{{ form }}
```

Run your app and go to *localhost:8000/accounts/signupaccount* .

And you will see a form with three fields, username, password1 and password2 (fig. 18.1).

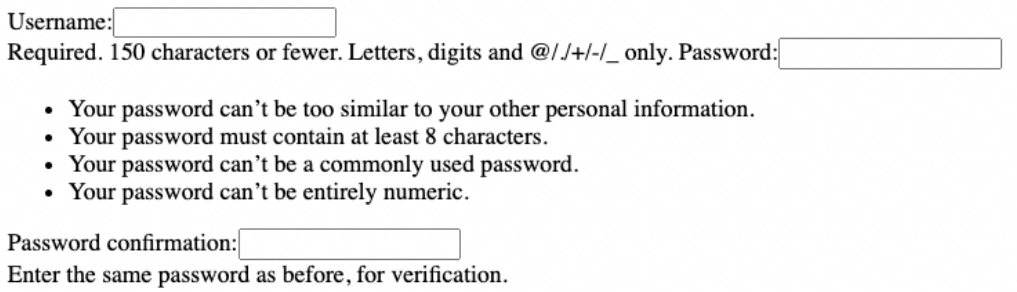


Figure 18.1

The form verifies that password1 and password2 matches. The look of the form has much to be desired though.

Let ' s improve the styling by having *signupaccount.html* extend *base.html* . We will also wrap the fields in a form tag and have a submit button. Replace the *signupaccount.html* with:

```
{% extends 'base.html' %}
{% block content %}
<div class="card mb-3">
 <div class="row g-0">
  <div>
   <div class="card-body">
    <h5 class="card-title">Sign Up</h5>
    <p class="card-text">
     <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit" class="btn btn-primary">
       Sign Up
      </button>
     </form>
    </p>
   </div>
```

```
  </div>
 </div>
</div>
{% endblock content %}
```

Code Explanation

The form doesn ' t have an *action* meaning that it submits to the same page. We will show why this is so later. Note also that the form method is ' POST ' , i.e. when the form submits, it sends a POST request since we are sending data to the server. This is different from the sign up mailing list form earlier where the form method is ' GET ' because we receive data from the signup form.

' POST ' keeps the submitted information hidden from the url. ' GET ' in contrast has the submitted data in the url eg *http://localhost:8000/signup/?email=greg%40greglim.com* . Because username and password are sensitive information, we want them hidden. A POST request will not put the information in the url.

In general, we use POST requests for creation and GET requests for retrieval. There are also other requests like *update* , *delete* and *put* but these are more important for creating APIs.

There is a line *{% csrf_token %}* . Django provides this to protect our form from cross-site request forgery (CSRF) attacks. You should use it for all your Django forms.

To output our form data, we use *{{ form.as_p }}* which renders it within paragraph <p> tags.

When you run your site, the form will now look something like (fig. 18.2):

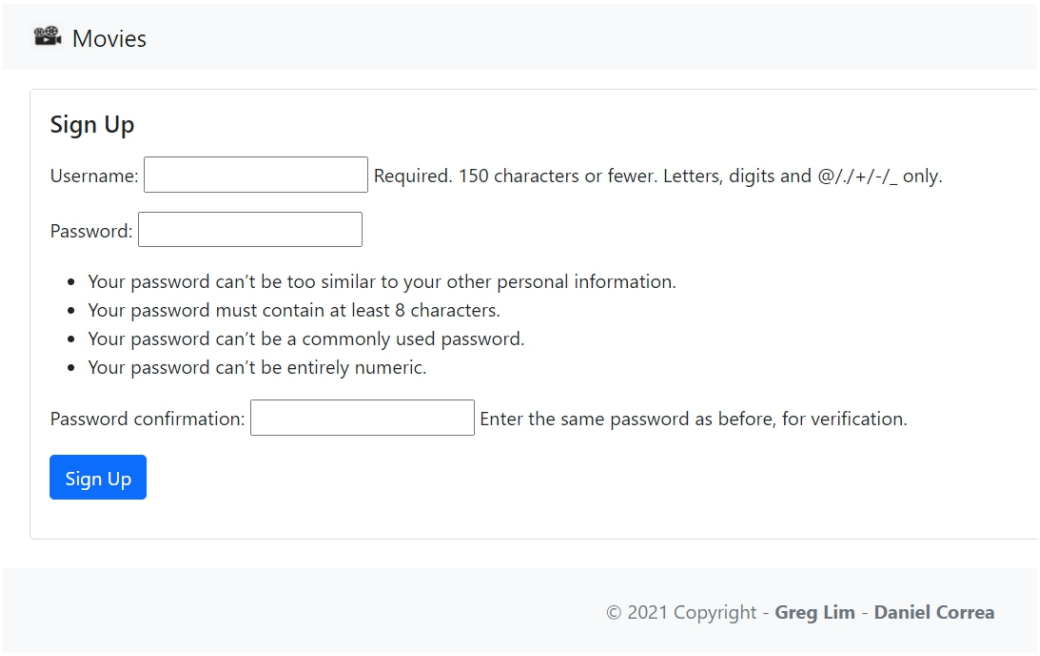


Figure 18.2

In the next chapter, we see how to handle the request and create a user in admin when the user submits the signup form.

Chapter 19: Creating a User

When the user submits the signup form, we will have to handle the request and create a user in admin. To do so, add the following in */accounts/views.py* :

```
from django.shortcuts import render
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User
from django.contrib.auth import login
from django.shortcuts import redirect

def signupaccount(request):
    if request.method == 'GET':
        return render(request, 'signupaccount.html',
                      {'form':UserCreationForm})
    else:
        if request.POST['password1'] == request.POST['password2']:
            user = User.objects.create_user(request.POST['username'],
                password=request.POST['password1'])
            user.save()
            login(request, user)
            return redirect('home')
```

Code Explanation

```
def signupaccount(request):
    if request.method == 'GET':
        return render(request, 'signupaccount.html',
                      {'form':UserCreationForm})
    else:
        …
```

In *def signupaccount* , we first check if the request received is a GET or POST request. If it is a GET request, it means that it's a user navigating to the sign up form via the url *localhost:8000/accounts/signupaccount* , in which case we simply send them to *signupaccount.html* with the form.

But if it's a POST request, it means that it's a form submission to create a

new user. And we move to the *else* block to create a new user.

```
else:
    if request.POST['password1'] == request.POST['password2']:
```

In the *else* block, we ensure that the password entered into *password1* and *password2* are the same before going on to create the user. What's *password1* and *password2* ? If you look at the 'View Page Source' markup for the form, you will see that the name for the password field is *password1* and the name for the password confirmation field is *password2* . So we first ensure that the password and confirm password values are the same before proceeding.

```
            user = User.objects.create_user(request.POST['username'],
                password=request.POST['password1'])
```

We then retrieve the data entered into the *username* field (*request.POST['username']*) and *password* field (*password=request.POST['password1']*). We pass in the data into *User.objects.create_user* which helps us create the user object. But where did the *User* model come from? We did not create it. The *User* model is provided by Django's Auth app (*from django.contrib.auth.models import User*) which has the *User* model in the database setup for us. If you recall, in Admin, we have Users (screenshot) which contain our superuser account created for us when we ran *python3 manage.py createsuperuser* .

```
            user.save()
```

user.save() actually inserts the new user into the database. The newly added user will show up in *Users* in Admin.

```
            login(request, user)
            return redirect('home')
```

After creating the user, we then login with the new user. That is, after someone signs up, we automatically log them in and redirect them to the home page.

Running Your App

Run your app now and go to *localhost:8000/accounts/signupaccount* . Create a user and you will see the new user added to the Admin.

Error Handling

But what happens if *password1* doesn't match *password2* ? To handle such an error, we add the below *else* block in **bold** :

```
…
def signupaccount(request):
    if request.method == 'GET':
        return render(request, 'signupaccount.html',
                      {'form':UserCreationForm})
    else:
        if request.POST['password1'] == request.POST['password2']:
            …
            return redirect('home')
        else:
            return render(request, 'signupaccount.html',
            {'form':UserCreationForm, 'error':'Passwords do not match'})
```

If the passwords don't match, we render the user back to *signupaccount.html* and also pass in an error message 'Passwords do not match'.

accounts/templates/signupaccount.html

We then render the error message into *signupaccount.html* with:

```
…
{% extends 'base.html' %}
{% block content %}
<div class="card mb-3">
  <div class="row g-0">
    <div>
      <div class="card-body">
        <h5 class="card-title">Sign Up</h5>
        {% if error %}
        <div class="alert alert-danger mt-3" role="alert">
          {{ error }}
        </div>
```

```
    {% endif %}
    <p class="card-text">
    …
```

Code Explanation

```
    {% if error %}
```

First, we only show the error message if it exists.

```
    <div class="alert alert-danger mt-3" role="alert">
      {{ error }}
    </div>
```

We render the error message in a bootstrap *alert* component so that a user can better notice the error and take the necessary corrective action (fig. 19.1).

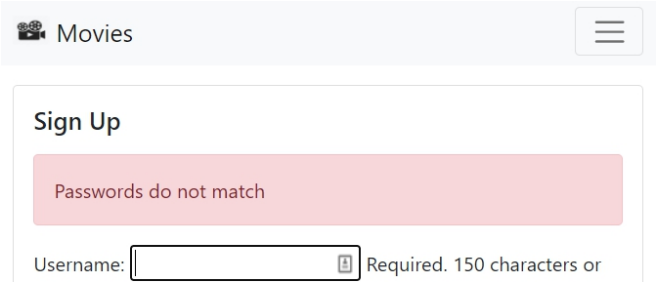


Figure 19.1

If a username already exists

We have illustrated an example of how to handle errors that arise when the form is populated incorrectly. You can implement validation of other form data errors for e.g. if the password length is less than eight characters.

There can be other kinds of errors which are identified only from the database. For example, if the user signs up with a username that already exists in the database. To catch such an error that is thrown by the database, we have to use *try* and *except* as shown in **bold** below:

```
…
from django.shortcuts import redirect
from django.db import IntegrityError
```

```
def signupaccount(request):
    if request.method == 'GET':
        return render(request, 'signupaccount.html',
                      {'form':UserCreationForm})
    else:
        if request.POST['password1'] == request.POST['password2']:
            try:
                user = User.objects.create_user(request.POST['username'],
                  password=request.POST['password1'])
                user.save()
                login(request, user)
                return redirect('home')
            except IntegrityError:
                return render(request, 'signupaccount.html',
                {'form':UserCreationForm,
                'error':'Username already taken. Choose new username.'})
        else:
            …
```

We import *IntegrityError* and using *try -except* , we catch *IntegrityError* when it is thrown (in the case when a username already exists – fig. 19.2)

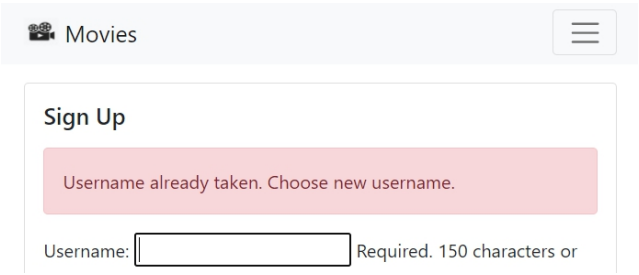


Figure 19.2

Customizing UserCreationForm

The *UserCreationForm* currently shows quite a lot of extra help text (included by default) which are cluttering our form. We can actually customize the *UserCreationForm* which is a big topic on its own. Here, we will simply remove the default help text.

To customize the form, we have to create a new class which extends

UserCreationForm . In */accounts/* , create a new file called *forms.py* and fill it in with:

```
from django.contrib.auth.forms import UserCreationForm


class UserCreateForm(UserCreationForm):
    def __init__(self, *args, **kwargs):
        super(UserCreateForm, self).__init__(*args, **kwargs)

        for fieldname in ['username', 'password1', 'password2']:
            self.fields[fieldname].help_text = None
            self.fields[fieldname].widget.attrs.update(
                                    {'class': 'form-control'})
```

Code Explanation

```
class UserCreateForm(UserCreationForm):
```

We created a new form *UserCreateForm* which extends from *UserCreationForm* .

```
    def __init__(self, *args, **kwargs):
        super(UserCreateForm, self).__init__(*args, **kwargs)
```

In the form's constructor, *def __init__(self, *args, **kwargs):* , we call the *super* method.

```
        for fieldname in ['username', 'password1', 'password2']:
            self.fields[fieldname].help_text = None
            self.fields[fieldname].widget.attrs.update(
                                    {'class': 'form-control'})
```

We set the *help_text* of the form's fields to 'None' to remove them, and we set for each form field a class attribute to use a Bootstrap class.

Using the UserCreateForm

Lastly, in */accounts/views.py* , change the following in **bold** to use *UserCreateForm* instead of *UserCreationForm* :

```
from django.shortcuts import render
```

```
~~from django.contrib.auth.forms import UserCreationForm~~
from .forms import UserCreateForm
from django.contrib.auth.models import User

def signupaccount(request):
    if request.method == 'GET':
        return render(request, 'signupaccount.html',
                      {'form': UserCreateForm })
    else:
        if request.POST['password1'] == request.POST['password2']:
            try:
                user = User.objects.create_user(request.POST['username'],
                    password=request.POST['password1'])
                user.save()
                login(request, user)
                return redirect('home')
            except IntegrityError:
                return render(request, 'signupaccount.html',
                 {'form': UserCreateForm ,
                 'error':'Username already taken. Choose new username.'})
        else:
                return render(request, 'signupaccount.html',
            {'form': UserCreateForm , 'error':'Passwords do not match'})
```

When you go to the sign up account form, the help text will not be there and the input will have a new style (fig. 19.3).

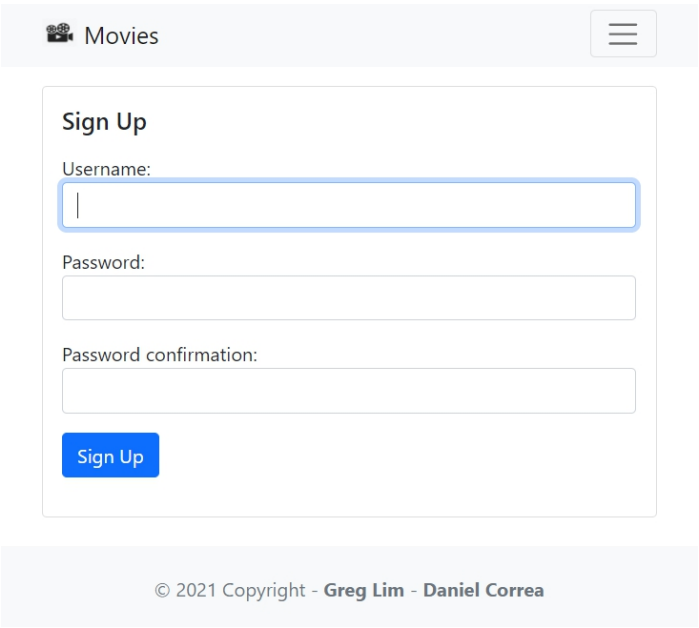


Figure 19.3

In the next chapter, we will show if a user is logged in on the navbar.

* You can obtain the source code of the completed project at www.greglim.co/p/django.

Chapter 20: Showing if a User is Logged In

After a user has signed up and logged in, we are still showing the Login and Sign Up buttons in the navbar (fig. 20.1).

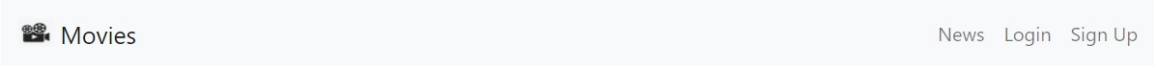


Figure 20.1

For logged in users, we should be hiding these buttons and show the Logout button instead. To do so, let's go to our base template. Remember that our base template is the starting point for everything and we extend it to the different views.

In *base.html* , add the codes in **bold** :

```
…
<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
 …
  <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
    <div class="navbar-nav ms-auto">
      <a class="nav-link" href="{% url 'news' %}">News</a>
      {% if user.is_authenticated %}
        <a class="nav-link" href="#">Logout ({{ user.username }})</a>
      {% else %}
        <a class="nav-link" href="#">Login</a>
        <a class="nav-link" href="#">Sign Up</a>
      {% endif %}
    </div>
  </div>
 …
```

Code Explanation

```
      {% if user.is_authenticated %}
        <a class="nav-link" href="#">Logout ({{ user.username }})</a>
```

```
    {% else %}
```

Notice that we have a *user* object which Django automatically provides (via the installed *auth* app) and passes in for us. The User object contains the username, password, email, first_name, and last_name properties. Additionally, we can check if a user is logged in with *{% if user.is_authenticated % }*

If user is authenticated, we render a logout button with the corresponding username *{{ user.username } }* . Else, it means the user is not logged-in and we show the login and signup buttons.

In the next chapter, we will implement the logout functionality.

CHAPTER 21: LOGOUT

Create the logout path in */accounts/urls.py* :

```
…
urlpatterns = [
    path('signupaccount/', views.signupaccount, name='signupaccount'),
     path('logout/', views.logoutaccount, name='logoutaccount'),
]
```

And in */accounts/views.py* , implement the *logoutaccount* function,

```
…
from django.contrib.auth.models import User
from django.contrib.auth import login , logout
…

def logoutaccount(request):
    logout(request)
    return redirect('home')
```

We simply call logout and redirect back to the home page.

We then need to have the <a href> in the logout button call the logout path. We also have the signup button call the *signupaccount* path. So in *base.html* , add in **bold** :

```
…
<nav class="navbar navbar-expand-lg navbar-light bg-light mb-3">
  …
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
      <div class="navbar-nav ms-auto">
        <a class="nav-link" href="{% url 'news' %}">News</a>
        {% if user.is_authenticated %}
          <a class="nav-link" href=" {% url 'logoutaccount' %} ">
            Logout ({{ user.username }})
             </a>
        {% else %}
          <a class="nav-link" href="#">Login</a>
```

```
        <a class="nav-link" href=" {% url 'signupaccount' %} ">
          Sign Up
        </a>
      {% endif %}
    </div>
  </div>
…
```

When you are logged in, you can now log out by clicking on the logout button.

Chapter 22: Log In

Having implemented sign up and log out, let's now implement log in. Create a login path in *accounts/urls.py* ,

```
…
urlpatterns = [
    path('signupaccount/', views.signupaccount, name='signupaccount'),
    path('logout/', views.logoutaccount, name='logoutaccount'),
     path('login/', views.loginaccount, name='loginaccount'),
]
```

And in */accounts/views.py* , implement *loginaccount* with:

```
…
from django.contrib.auth.models import User
from django.contrib.auth.forms import AuthenticationForm
from django.contrib.auth import login, logout , authenticate
…

def loginaccount(request):
    if request.method == 'GET':
        return render(request, 'loginaccount.html',
                      {'form':AuthenticationForm})
    else:
        user = authenticate(request, username=request.POST['username'],
                            password=request.POST['password'])
        if user is None:
            return render(request,'loginaccount.html',
                          {'form': AuthenticationForm(),
                'error': 'username and password do not match'})
        else:
            login(request,user)
            return redirect('home')
```

Code Explanation

```
def loginaccount(request):
    if request.method == 'GET':
        return render(request, 'loginaccount.html',
                      {'form':AuthenticationForm})
```

loginaccount will be similar to *signupaccount* . We first handle the case where the request is a GET request, i.e. the user clicks on *Login* on the navbar, and we render *loginaccount.html* . We then pass in the *AuthenticationForm* . Much like the *UserCreationForm* for signup, Django provides the *AuthenticationForm* to quickly get a login form up and running.

```
else:
    user = authenticate(request, username=request.POST['username'],
                                password=request.POST['password'])
      if user is None:
            return render(request,'loginaccount.html',
                        {'form': AuthenticationForm(),
             'error': 'username and password do not match'})
      else:
        login(request,user)
        return redirect('home')
```

If the request type is not GET (user submits the login form and sends a POST request), we proceed to authenticate the user with the values he entered in the username and password fields.
If the user returned from *authenticate* is *None* , i.e. we are unable to find an existing user with the supplied username/password, we return the user to *loginaccount.html* with the error 'username and password do not match' (fig. 22.1).

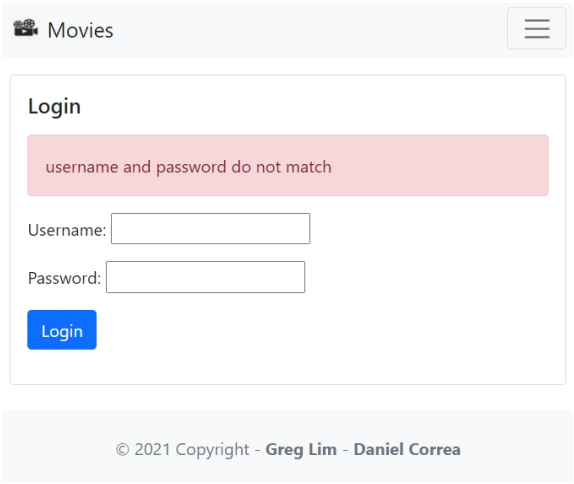


Figure 22.1

Else, it means the authentication is successful and we log in the user and redirect her to the home page.

/accounts/templates/loginaccount.html

Create the new file *accounts/templates/loginaccount.html* , and copy the markup from *accounts/templates/signupaccount.html* and change the labeling from ‘Sign Up’ to ‘Login’:

```
{% extends 'base.html' %}
{% block content %}
<div class="card mb-3">
  <div class="row g-0">
    <div>
      <div class="card-body">
        <h5 class="card-title"> Login </h5>
        {% if error %}
        <div class="alert alert-danger mt-3" role="alert">
          {{ error }}
        </div>
        {% endif %}
        <p class="card-text">
          <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <button type="submit" class="btn btn-primary"> Login </button>
          </form>
        </p>
      </div>
    </div>
  </div>
</div>
{% endblock content %}
```

Finally, in *base.html* , we set the *href* for *loginaccount* :

```
…
        {% if user.is_authenticated %}
          <a class="nav-link" href="{% url 'logoutaccount' %}">
            Logout ({{ user.username }})
            </a>
        {% else %}
```

```
        <a class="nav-link" href=" {% url 'loginaccount' %} ">Login</a>
        <a class="nav-link" href="{% url 'signupaccount' %}">
          Sign Up
        </a>
      {% endif %}
…
```

Our navbar is now complete and fully functioning. For users not logged in, the navbar will show the login and sign up links. When a user logs in, the navbar will show only the logout button. In the next chapter, we will implement letting logged-in users post reviews for movies.

Chapter 23: Letting Users Post Movie Reviews

We will now implement letting logged-in users post reviews for movies. We have to first create a Review model.

In *movie/models.py* , add the following to define a Review model:

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

class Review(models.Model):
    text = models.CharField(max_length=100)
    date = models.DateTimeField(auto_now_add=True)
    user = models.ForeignKey(User,on_delete=models.CASCADE)
    movie = models.ForeignKey(Movie,on_delete=models.CASCADE)
    watchAgain = models.BooleanField()

    def __str__(self):
        return self.text
```

Code Explanation

```
    text = models.CharField(max_length=100)
```

The text field stores the review text.

```
    date = models.DateTimeField(auto_now_add=True)
```

For review date, we specify auto_now_add=Tru e . That is, when someone creates this object, the current datetime will be automatically filled in. Note that this makes the field non-editable. Once the datetime is set, it is fixed.

```
    user = models.ForeignKey(User,on_delete=models.CASCADE)
    movie = models.ForeignKey(Movie,on_delete=models.CASCADE)
```

For the user and movie field, we are using a ForeignKey which allows for a many-to-one relationship. This means that a user can create multiple reviews.

A movie similarly can have multiple reviews.
For user, the reference is to the built-in User model that Django provides for authentication. For all many-to-one relationships such as a ForeignKey, we must also specify an *on_delete* option. That is, when you remove a user or movie for instance, its associated reviews will be deleted as well. Note that this does not apply the other direction i.e. when you remove a review, the associated movie and user still remains.

```
    watchAgain = models.BooleanField()
```

Lastly, we have a boolean property *watchAgain* for users to indicate if they will watch the movie again.

Registering to Admin

To have our Review model appear in the admin dashboard, remember that we have to register it by adding the following in **bold** into */movie/admin.py* :

```
from django.contrib import admin
from .models import Movie , Review


admin.site.register(Movie)
admin.site.register(Review)
```

In Terminal, make the migration for the new model with:

```
python3 manage.py makemigrations
```

And then apply the changes to the sqlite3 database with:

```
python3 manage.py migrate
```

In the next chapter, we will see how to let users post reviews on movies from the site.

Chapter 24: Creating a Review

We have seen how to create model objects from the admin e.g. creating a *movie* object. But how do we allow users to create their own objects e.g. let users post a review from the site? After all, not everyone should have access to the admin panel.

Let's create a page for them to do so. We first create a path in */movie/urls.py* :

```
from django.urls import path
from . import views

urlpatterns = [
    path('<int:movie_id>', views.detail, name='detail'),
     path('<int:movie_id>/create', views.createreview,
        name='createreview'),
]
```

/movie/views.py

And in */movie/views.py* , add the following *def createreview* :

```
…
from django.shortcuts import get_object_or_404 , redirect
from .models import Movie , Review
from .forms import ReviewForm
…

def createreview(request, movie_id):
    movie = get_object_or_404(Movie,pk=movie_id)
   if request.method == 'GET':
        return render(request, 'createreview.html',
                {'form':ReviewForm(), 'movie': movie})
   else:
      try:
         form = ReviewForm(request.POST)
         newReview = form.save(commit=False)
         newReview.user = request.user
```

```
        newReview.movie = movie
        newReview.save()
        return redirect('detail', newReview.movie.id)
    except ValueError:
            return render(request, 'createreview.html',
        {'form':ReviewForm(),'error':'bad data passed in'})
```

Code Explanation

```
    movie = get_object_or_404(Movie,pk=movie_id)
```

We first get the movie object from the database.

```
    if request.method == 'GET':
        return render(request, 'createreview.html',
                {'form':ReviewForm(), 'movie': movie})
```

When we receive a GET request, it means that a user is navigating to the create review page and we render *createreview.html* and pass in the review form for the user to create the review. We will later show how to create the Review Form.

```
    else:
        try:
```

When user submits the *createreview* form, this function will receive a POST request, and we enter the *else* clause.

```
            form = ReviewForm(request.POST)
```

We retrieve the submitted form from the request.

```
            newReview = form.save(commit=False)
```

We create and save a new review object from the form's values but do not yet put it into the database *(commit=False)* because we want to specify the user and movie relationships for the review.

```
            newReview.user = request.user
            newReview.movie = movie
            newReview.save()
            return redirect('detail', newReview.movie.id)
```

Finally, we specify the user and movie relationships for the review and save the review into the database. We then redirect the user back to the movie's detail page.

```
    except ValueError:
            return render(request, 'createreview.html',
        {'form':ReviewForm(),'error':'bad data passed in'})
```

If there's any error with the passed-in data, we render *createreview.html* again and pass in an error message. Let's next create the *createreview.html* page.

/movie/createreview.html

In */movie/templates* , create a new file *createreview.html* with the below code:

```
{% extends 'base.html' %}
{% block content %}
<div class="card mb-3">
  <div class="row g-0">
    <div>
      <div class="card-body">
        <h5 class="card-title">Add Review for {{ movie.title }}</h5>
        {% if error %}
        <div class="alert alert-danger mt-3" role="alert">
          {{ error }}
        </div>
        {% endif %}
        <p class="card-text">
          <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <button type="submit" class="btn btn-primary">
              Add Review
            </button>
          </form>
        </p>
      </div>
```

```
    </div>
  </div>
</div>
{% endblock content %}
```

As you can see, *createreview.html* is very similar to the other pages. So let ' s move ahead and see how to create the review form.

ReviewForm

To create the review form, we can make use of *ModelForm* provided by Django to automatically create forms from models.
In */movie* , create the file *forms.py* and fill it with the following code:

```
from django.forms import ModelForm, Textarea
from .models import Review

class ReviewForm(ModelForm):
    def __init__(self, *args, **kwargs):
        super(ModelForm, self).__init__(*args, **kwargs)
        self.fields['text'].widget.attrs.update(
                        {'class': 'form-control'})
        self.fields['watchAgain'].widget.attrs.update(
                            {'class': 'form-check-input'})

    class Meta:
        model = Review
          fields = ['text','watchAgain']
        labels = {
            'watchAgain': ('Watch Again')
        }
        widgets = {
            'text': Textarea(attrs={'rows': 4}),
        }
```

Code Explanation

```
class ReviewForm(ModelForm):
```

We need to inherit from ModelForm.

```
def __init__(self, *args, **kwargs):
    super(ModelForm, self).__init__(*args, **kwargs)
    self.fields['text'].widget.attrs.update(
                    {'class': 'form-control'})
    self.fields['watchAgain'].widget.attrs.update(
                        {'class': 'form-check-input'})
```

Similar as what we did with the *UserCreationForm* , we set some Bootstrap classes for our form fields.

```
class Meta:
    model = Review
    fields = ['text','watchAgain']
```

We then specify which model the form is for and the fields we want in the form. In our case, our review form will need just the *text* and *watchAgain* fields. If you recall, in our Review model:

```
class Review(models.Model):
    text = models.CharField(max_length=100)
    date = models.DateTimeField(auto_now_add=True)
    user = models.ForeignKey(User,on_delete=models.CASCADE)
    movie = models.ForeignKey(Movie,on_delete=models.CASCADE)
    watchAgain = models.BooleanField()
```

date is auto-populated, user and movie are already provided. Thus, we need only user to input the *text* and *watchAgain* fields in the form.

```
    labels = {
        'watchAgain': ('Watch Again')
    }
```

We have a labels object where we can create custom labels for each of our fields. For e.g., we want to display ‘ Watch Again ’ instead of *watchAgain* (our users are not programmers!).

```
    widgets = {
        'text': Textarea(attrs={'rows': 4}),
    }
```

By default, a CharField is displayed as an input text. We override this default field (with the use of widgets) to have a Textarea for our text field.

For more information about ModelForms, you can check out the Django documentation at https://docs.djangoproject.com/en/3.2/topics/forms/modelforms/.

/movie/templates/detail.html

Finally, we render a ‘ Add Review ’ button in the movie details page with the following codes in **bold** :

```
…
    <div class="card-body">
      <h5 class="card-title">{{ movie.title }}</h5>
      <p class="card-text">{{ movie.description }}</p>
      <p class="card-text">
        {% if movie.url %}
          <a href="{{ movie.url }}" class="btn btn-primary">
            Movie Link
          </a>
        {% endif %}
        {% if user.is_authenticated %}
              <a href="{% url 'createreview' movie.id %}"
          class="btn btn-primary">
          Add Review
        </a>
          {% endif %}
      </p>
    </div>
…
```

Notice that we have enclosed the ‘ Add Review ’ link in an **if user.is_authenticated** block. This is to ensure that we only allow logged-in users to add a review. Users who are not logged in will not see the ‘ Add Review ’ link.

Running your App

Log in, go to a movie and click ‘ Add Review ’ (fig. 24.1).

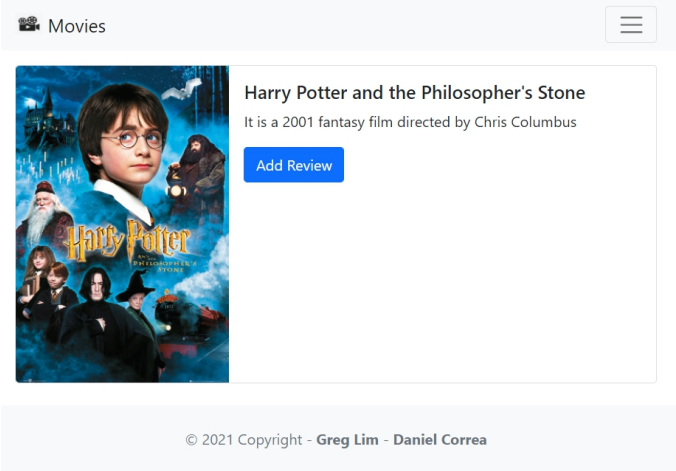


Figure 24.1

You will see the Review Form that Django automatically generated for you (fig. 24.2)!

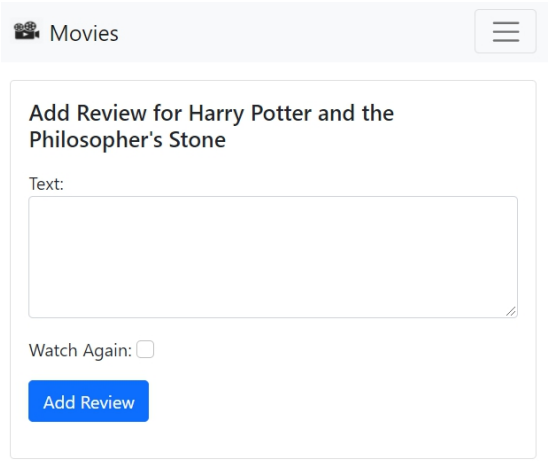


Figure 24.2

After adding the review, you can check the Admin where it will be reflected. In the next chapter, we will see how to list reviews in the movie details page.

Chapter 25: Listing Reviews

Now, we want to list a movie's reviews in the movie details page (fig. 25.1).

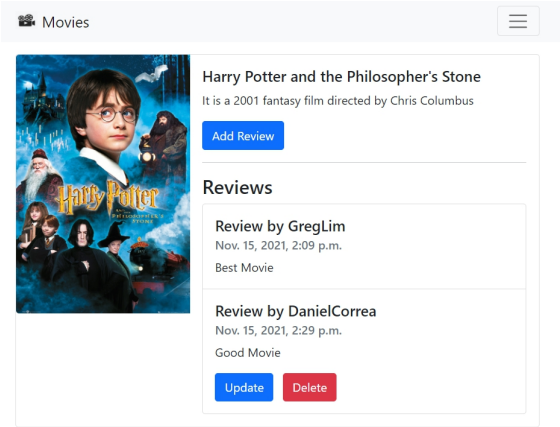


Figure 25.1

To do so, in */movie/views.py* , in *def detail,* add the codes in **bold** :

```
def detail(request, movie_id):
    movie = get_object_or_404(Movie,pk=movie_id)
    reviews = Review.objects.filter(movie = movie)
     return render(request, 'detail.html',
                {'movie':movie , 'reviews': reviews} )
```

Code Explanation

```
    reviews = Review.objects.filter(movie = movie)
```

Using the *filter* function, we retrieve reviews for a particular movie only.

```
    return render(request, 'detail.html',
                {'movie':movie , 'reviews': reviews} )
```

We then pass in *reviews* to *detail.html* .

/movie/templates/detail.html

We list the reviews under the Movie's card component by adding the section in **bold** :

```
...
```

```
    <div class="card-body">
      <h5 class="card-title">{{ movie.title }}</h5>
      <p class="card-text">{{ movie.description }}</p>
      <p class="card-text">
          …
      </p>
       <hr />
      <h3>Reviews</h3>
      <ul class="list-group">
        {% for review in reviews %}
        <li class="list-group-item pb-3 pt-3">
          <h5 class="card-title">
            Review by {{ review.user.username }}
          </h5>
          <h6 class="card-subtitle mb-2 text-muted">
            {{ review.date }}
          </h6>
          <p class="card-text">{{ review.text }}</p>
              {% if user.is_authenticated and user == review.user %}
          <a class="btn btn-primary me-2">Edit</a>
          <a class="btn btn-danger">Delete</a>
          {% endif %}
        </li>
      {% endfor %}
      </ul>
    </div>
…
```

Code Explanation

```
    <ul class="list-group">
      {% for review in reviews %}
      <li class="list-group-item pb-3 pt-3">
        ...
      </li>
    {% endfor %}
    </ul>
```

Using a *for-loop* , we render a Bootstrap List Group Item component for each

review (https://getbootstrap.com/docs/5.1/components/list-group/).

```
<h5 class="card-title">
  Review by {{ review.user.username }}
</h5>
<h6 class="card-subtitle mb-2 text-muted">
  {{ review.date }}
</h6>
<p class="card-text">{{ review.text }}</p>
```

We render the username, the review date and the review text.

```
    {% if user.is_authenticated and user == review.user %}
<a class="btn btn-primary me-2" href="#">Update</a>
<a class="btn btn-danger" href="#">Delete</a>
{% endif %}
```

We also check if a user is logged-in, and if a review belongs to the user, render the update and delete link to allow her to update/delete it. Else, we hide the update/delete links, i.e. a user can only update/delete reviews they have posted. They can't do so for others' reviews.

Running Your App

In the movie details page, you will be able to see the reviews for a movie now. If you are logged-in, you can see the update and delete buttons for reviews you posted (fig. 25.2).

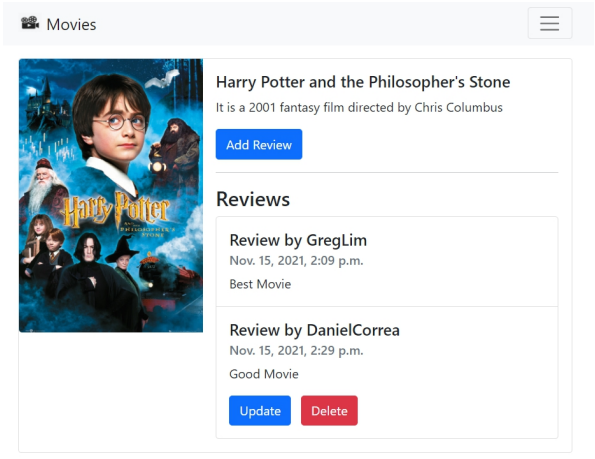


Figure 25.2

When you log out, you can't see them anymore. In the next few chapters, we will go on to implement the *updatereview* and *deletereview* functions.

Chapter 26: Updating a Review

We create a url path to update a review in */movie/urls.py* :

```
…

urlpatterns = [
    path('<int:movie_id>', views.detail, name='detail'),
     path('<int:movie_id>/create', views.createreview,
        name='createreview'),
     path('review/<int:review_id>', views.updatereview,
        name='updatereview'),
]
```

The path takes in the review id (review's primary key). E.g. *localhost:8000/movie/review/2*

In */movie/views.py* , we then add *def updatereview* :

```
…

def updatereview(request, review_id):
    review = get_object_or_404(Review,pk=review_id,user=request.user)
   if request.method =='GET':
       form = ReviewForm(instance=review)
         return render(request, 'updatereview.html',
                  {'review': review,'form':form})
   else:
       try:
          form = ReviewForm(request.POST, instance=review)
          form.save()
               return redirect('detail', review.movie.id)
       except ValueError:
          return render(request, 'updatereview.html',
           {'review': review,'form':form,'error':'Bad data in form'})
```

Code Explanation

```
    review = get_object_or_404(Review,pk=review_id,user=request.user)
```

We first retrieve the review object with the review id. We also supply the logged-in user to ensure that other users can't access the review e.g. if they manually enter the url path in the browser. Only the user who created this review can update/delete it.

```
if request.method =='GET':
    form = ReviewForm(instance=review)
      return render(request, 'updatereview.html',
                {'review': review,'form':form})
```

If the request type is GET, it means they navigated to the page from the movie details page. We thus render the ReviewForm we used previously in creating a review, but this time, we pass in the review object into the form so that the form's fields will be populated with the object's values, ready for the user to edit. See how Django's *ModelForm* saves us so much work!

```
else:
    try:
        form = ReviewForm(request.POST, instance=review)
        form.save()
            return redirect('detail', review.movie.id)
```

In the *else* block, i.e. the request type is POST which means the user is trying to submit the update form. We retrieve the values from the form and do a *form.save* () to update the existing review. We then redirect back to the movie details page.

```
    except ValueError:
        return render(request, 'updatereview.html',
         {'review': review,'form':form,'error':'Bad data in form'})
```

If there is a problem with the content the user has provided, we catch it with the *ValueError* exception.

/movie/templates/updatereview.html

We create a new file */movie/templates/updatereview.html* and fill it with the following:

```
{% extends 'base.html' %}
{% block content %}
```

```
<div class="card mb-3">
  <div class="row g-0">
    <div>
      <div class="card-body">
        <h5 class="card-title">
                  Update Review for {{ review.movie.title }}
                </h5>
        {% if error %}
        <div class="alert alert-danger mt-3" role="alert">
          {{ error }}
        </div>
        {% endif %}
        <p class="card-text">
          <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <button type="submit" class="btn btn-primary">
              Update Review
            </button>
          </form>
        </p>
      </div>
    </div>
  </div>
</div>
{% endblock content %}
```

Again, the markup above is similar to the other template files. See how Django greatly simplifies template and form creation for us!

/movie/templates/detail.html

Lastly, back in *detail.html* , we add the ‘updatereview’ url to the update button.

```
…
                {% if user.is_authenticated and user == review.user %}
                <a class="btn btn-primary me-2"
            href=" {% url 'updatereview' review.id %} ">
```

```
            Update
          </a>
          <a class="btn btn-danger" href="#">Delete</a>
          {% endif %}
...
```

Running your app

When we run our app and try to update a review, the form will appear, and it will be filled with the existing review's values (fig. 26.1).

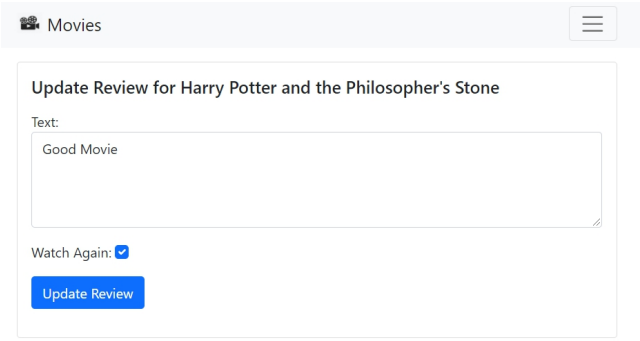


Figure 26.1

Update the review with new values and upon submitting the form, the updated review will be reflected in the movie details page.

Chapter 27: Deleting a Review

Having implemented update of a review, let ' s now implement deleting of a review. Create a new path to delete a review in */movie/urls.py* (you should be familiar with this by now):

```
…

urlpatterns = [
    path('<int:movie_id>', views.detail, name='detail'),
     path('<int:movie_id>/create', views.createreview,
         name='createreview'),
     path('review/<int:review_id>', views.updatereview,
         name='updatereview'),
     path('review/<int:review_id>/delete', views.deletereview,
          name='deletereview')
]
```

And in */movie/views.py* , add *def deletereview* :

```
…

def deletereview(request, review_id):
    review = get_object_or_404(Review, pk=review_id, user=request.user)
    review.delete()
     return redirect('detail', review.movie.id)
```

You can see that *deletereview* is quite straightforward. We get the *review* object and call its *delete* method. Like for update, we supply the logged-in user to ensure only the user who created this review can delete it. We then redirect back to the movie ' s detail page.

Back in the listing of reviews in the movie details page (*/movie/templates/detail.html*), we now add the *deletereview* url:

```
…
                {% if user.is_authenticated and user == review.user %}
                <a class="btn btn-primary me-2"
            href="{% url 'updatereview' review.id %}">
            Update
```

```
        </a>
            <a class="btn btn-danger"
          href=" {% url 'deletereview' review.id %} ">
          Delete
        </a>
        {% endif %}
…
```

Running Your App

When you run your app now, log in and go to a specific movie, a user will be able to delete reviews they have posted.

Chapter 28: Authorization

We have implemented authentication where we allow users to sign up and log in. But we also need authorization which authorizes access to certain pages only to logged in users.

Currently, if a user manual enters the url to create a review e.g. *http://localhost:8000/movie/2/create* , they can still access the form. We should authorize access to creating/updating/deleting of reviews only to logged in users. We also authorize access to *logout* .

To do so, we import and add the *@login_required* decorator to the views which we want to authorize as shown in **bold** :

/movie/views.py

```
…
from .forms import ReviewForm
from django.contrib.auth.decorators import login_required
…

@login_required
def createreview(request, movie_id):
    …

@login_required
def updatereview(request, review_id):
    …

@login_required
def deletereview(request, review_id):
    …
```

/accounts/views.py

```
…
from django.db import IntegrityError
from django.contrib.auth.decorators import login_required
```

```
…

@login_required
def logoutaccount(request):
    …
```

We also have to add at the end of */moviereviews/settings.py* :

```
…

LOGIN_URL = 'loginaccount'
```

This redirects a user (who is not logged in) to the login page when they attempt to access an authorized page.

When you run your app now, ensure that you are logged out and go to the create review page e.g. *http://localhost:8000/movie/2/create* , you will be redirected to the login page.

* You can obtain the source code of the completed project at www.greglim.co/p/django.

Chapter 29: Deployment

Our project is currently running on our local machine. To have our site live on the real, public internet for the world to use, we need to deploy it onto an actual server. A popular way to do so is to deploy our Django project on PythonAnywhere. For one, it is free to use for small websites.

To get our code on to PythonAnywhere, we need our code to be on a code sharing site like GitHub or Bitbucket. In this chapter, we will use GitHub. If you are already familiar with uploading your code to GitHub, please skip the following section. Else, you can follow along.

GitHub and Git

In this section, we cover only the necessary steps to move your code onto GitHub. We will not cover all details about Git and GitHub (entire books have been written on these alone!).

Go to *github.com* and sign up for an account if you don't have one. To put your project on GitHub, you will need to create a repository for it to live in. Create a new repository by clicking on '+' on the top-right and select 'New repostiory' (fig. 29.1).

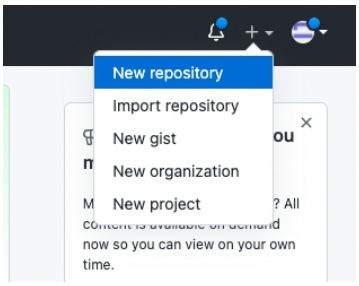


Figure 29.1

Give your repository a name e.g. *moviereviews* . Select the 'public' radio box and hit 'Create repository'.

We will begin to move our code onto GitHub. In your local machine's Terminal, ensure you have *git* installed by running: *git*

Now, what is Git? Git is a version control for projects and is very popular in the development world. It allows us to have save points (Git calls them

commits) in our code. If we make mistakes in our project at any point in time, we can go back to previous save points when the project is working. Git also allows multiple developers to work on the project together without worrying about one overwriting the code of another.

When you run *git* in the Terminal, if you see Git commands listed down, you have Git installed. If you don ' t see them, you will need to install Git. Visit the Git site (https://git-scm.com/) and follow the instructions to install Git. When Git is installed, you might need to close and re-open the Terminal and in it. Type *git* to ensure that it is installed.

In your project folder, enter:

```
git init
```

git init marks your folder as a Git project where you can begin to track changes. A hidden folder *.git* is added to the project folder. The *.git* folder stores all the objects and refs that Git uses and creates as part of your project ' s history.

Next, run:

```
git add -A
```

This adds everything in your project into the staging environment to prepare a snapshot before committing it to the official history. Go ahead to commit them by running:

```
git commit -m "first message"
```

This creates a save point in your code. You can identify different commits by the descriptive messages you provide. Next, we want to save our Git project on to GitHub.

In the repository page in GitHub, copy the *git remote add origin <origin path>* command and run it the Terminal. Eg:

```
git remote add origin https://github.com/greglim81/moviereviews.git
```

This creates a new connection record to the remote repository. To move the code from your local computer to GitHub, run:

```
git push -u origin main
```

This pushes the code to GitHub. When you reload the GitHub repository page, your project ' s code will be reflected there.

*Do note that there is much more to Git and GitHub. We have just covered the necessary steps to upload our code on to GitHub. With this, we have now gotten our code on GitHub. Next, we will clone our code on to PythonAnywhere.

Cloning our Code on to PythonAnywhere

The steps to deploy an existing Django project on PythonAnywhere can be found at https://help.pythonanywhere.com/pages/DeployExistingDjangoProject/ . But I will go through it with you here.

Now that we have our code on GitHub, we will have PythonAnywhere retrieve our code from there. First, create a beginner free account in PythonAnywhere here: https://www.pythonanywhere.com/registration/register/beginner/

In PythonAnywhere, click on ' New console ' -> ' Bash ' to access its Linux Terminal (fig. 29.2).

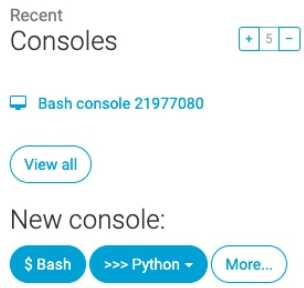


Figure 29.2

Back in your GitHub repository, click on ' Code ' and copy the URL to clone (fig. 29.3).

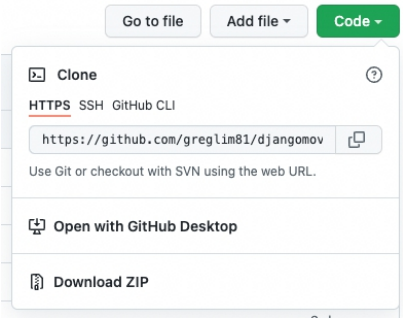


Figure 29.3

To clone, back in the PythonAnywhere Bash, run: *git clone <paste url>* , e.g.

```
git clone https://github.com/greglim81/djangomoviereviews.git
```

This will take all your code from the GitHub repository and clone it in PythonAnywhere. When the clone completes, you can do a ‘ ls ’ in bash and you will see the folder as what you see on your machine.

Virtual Environments

Virtual environments are used to create separate development environments for different projects with different requirements. For example, you can specify which version of Python and which libraries/modules you want installed in a particular virtual environment.

As an example, to create a virtual environment in the PythonAnywhere bash, we run:

```
mkvirtualenv -p python3.8 <environment name>
```

In the above, we have specified that we use Python 3.8 in the virtualenv. Whatever packages we install will always be there and independent of other virtualenvs.

If I ran:

```
mkvirtualenv -p python3.8 moviereviewsenv
```

I will see the name of the virtualenv in the Bash which means we are in the VE eg:

(**moviereviewsenv**) 00:08 ~ $

Back in our virtualenv, we need to install django and pillow (as what we have done in development). So run:

```
pip install django pillow
```

Setting up your Web app

At this point, you need to be ready with 3 pieces of information:

1. The path to your Django project's top folder -- the folder that contains "manage.py". A simple way to get this is, in your project folder in bash, type *pwd* . Eg: */home/greglim/djangomoviereviews*

2. The name of your project (the name of the folder that contains your settings.py), e.g. *moviereviews*

3. The name of your virtualenv, eg *moviereviewsenv*

Create a Web app with Manual Config

In your browser, open a new tab and go to the PythonAnywhere dashboard. Click on the ‘ Web ’ tab (fig. 29.4).

Figure 29.4

Click, ‘ Add a new web app ’ . Under ‘ Select a Python Web framework ’ , choose the ‘ Manual Configuration ’ (fig. 29.5).

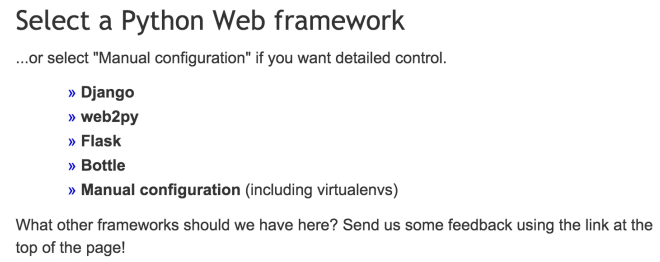


Figure 29.5

NOTE: Make sure you choose ‘Manual Configuration’, not the ‘Django’ option, that's for new projects only.

Select the right version of Python (the same one you used to create your virtualenv).

Enter your virtualenv name

Enter the name of your virtualenv in the *Virtualenv* section (fig. 29.6).

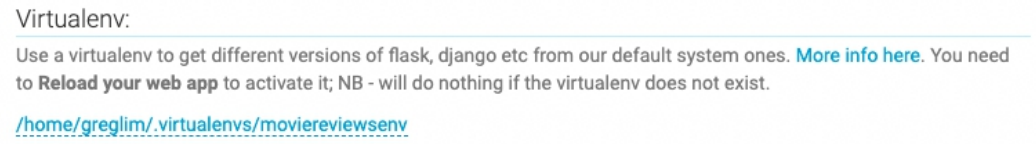


Figure 29.6

You can just use its short name "moviereviewsenv", and it will automatically complete to its full path in */home/username/.virtualenvs* .

Enter path to your code

Next, enter the path to your project folder in the *Code* section, both for ' Source code ' and ' Working directory ' (fig. 29.7):

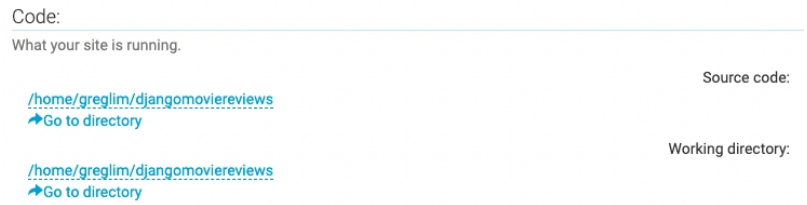


Figure 29.7

Edit your WSGI file

In the *wsgi.py* file inside the ' Code ' section (fig. 29.8 - not the one in your local Django project folder),

Figure 29.8

Click on the WSGI file link and it will take you to an editor where you can change it. Delete everything except the Django section and uncomment that section. Your WSGI file will look something like:

```
# ++++++++++ DJANGO ++++++++++
```

```
# To use your own Django app use code like this:
import os
import sys

path = '/home/greglim/djangomoviereviews/'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'moviereviews.settings'

# then:
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

Code Explanation

```
path = '/home/greglim/djangomoviereviews/'
```

Be sure to substitute the correct path to your project, the folder that contains *manage.py* ,

```
os.environ['DJANGO_SETTINGS_MODULE'] = 'moviereviews.settings'
```

Make sure you put the correct value for DJANGO_SETTINGS_MODULE and save the file.

Exiting and Entering a Virtualenv

At any point you want to get out of the virtualenv, run ‘ deactivate ’ in the Bash.
To get back in to a virtualenv, run: workon <virtualenv name>

If you forgot the name of a virtualenv, you can see the list of virtualenvs you have by going to the *.virtualenvs* folder:

```
cd .virtualenvs
ls
```

You can then see the list of virtualenvs.

Allowed Hosts

Next, we need to add to the allowed hosts in *settings.py* . Go to the ‘ Files ’ tab, navigate through the source code directory, and in *settings.py* , add:

```
ALLOWED_HOSTS = ['*']
```

Save the file. Then go to the ‘ Web ’ tab and hit the Reload button for your domain (fig. 29.9).

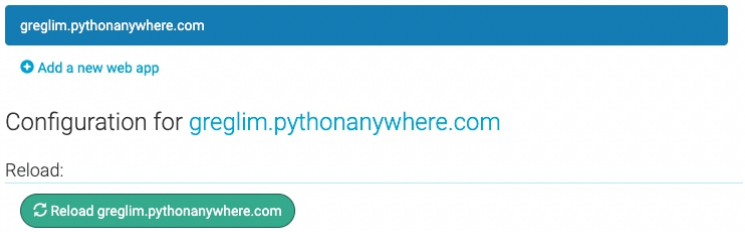


Figure 29.9

The ALLOWED_HOSTS settings represent which host/domain names our Django site can serve. This is a security measure to prevent HTTP Host header attacks. We used the wildcard Asterisk * to indicate all domains are acceptable. In your production projects, you can explicitly list which domains are allowed.

Go to your project ’ s url (it is the blue link in the previous image) and the home page should now show! We are almost there! But note that the *static* and *media* images are still not showing. Let ’ s fix that in the next section.

Static Files

Let ’ s fix the problem of our static and media images not showing. In *settings.py* , we have to add the following in **bold** and remove a line:

```
…
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR,'static')

# Default primary key field type
# https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field
```

```
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'


MEDIA_ROOT = os.path.join(BASE_DIR,'media')
MEDIA_URL = '/media/'

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
    'moviereviews/static/',
]
```

The STATIC_ROOT variable defines a central location we collect all static files into.

Save the file and back in bash console (inside the virtualenv), run:

```
python manage.py collectstatic
```

This command collects all your static files from each of your app folders (including the static files for the admin app) and from any other folders you specify in *settings.py* and copies them into STATIC_ROOT.
You need to re-run this command whenever you want to publish new versions of your static files.

Set up static files mapping

Next, set up a static files mapping to get our web servers to serve out your static files for you.

In the *Web* tab on the PythonAnywhere dashboard, under ‘ Static Files ’ , enter the same URL as STATIC_URL in the url section (typically, */static/*)

Enter the path from STATIC_ROOT into the path section (fig. 29.10 - the full path, including */home/username/* etc)

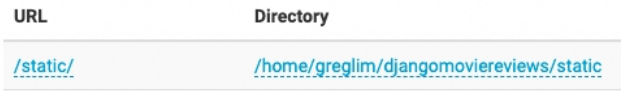


Figure 29.10

Then hit Reload and your static images should appear in your site now.

Set up media files mapping

Set up a similar static files mapping from MEDIA_URL to MEDIA_ROOT (fig. 29.11).

Figure 29.11

Hit Reload and your media images should appear in your site now (fig. 29.12).

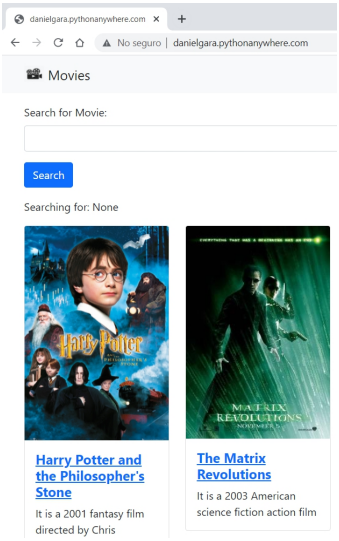


Figure 29.12

Set Debug to False

For local development, we have set DEBUG=True. This shows us the detailed error description and from which line of code it is resulting from. We should however hide these in production (we don ' t want to expose all our secrets out in the open) by setting DEBUG=False.

Thus, in the PythonAnywhere dashboard, go your project ' s *settings.py* and set DEBUG = False.

createsuperuser

Remember to create a superuser in your production environment (just as we did in our development environment) and keep the password secure.

.gitignore

We can ignore some files when uploading to GitHub. For e.g. the *__pycache__* directory which are created automatically when Django runs *.py* files. We should also ignore the *db.sqlite3* file as it might be accidentally pushed to production. And if you are using a Mac, we can ignore *.DS_Store* which stores information about folder settings on MacOS.

So the final *.gitignore* will look like:

```
__pycache__/
db.sqlite3
.DS_Store
```

Change *db.sqlite3* to MySQL or PostgresSQL

Our current database is set to SQLite which works fine for small projects. But when the project or the data size grows, we want to switch to other databases like MySQL or PostgreSQL.

PythonAnywhere allows usage of MySQL free. But for PostgreSQL, you will need to have a paid account.

To start using MySQL, you can refer to PythonAnywhere ' s brief and useful documentation (*https://help.pythonanywhere.com/pages/UsingMySQL/*).

Note: After you have setup MySQL or any other database, because it ' s a brand new database, you will have to create a new superuser (*python manage.py createsuperuser*) and run the makemigrations/migrate command:

```
python manage.py makemigrations
python manage.py migrate
```

Final Words

We have gone through quite a lot of content to equip you with the skills to create a Django full stack app. We have covered the major features of

Django, templates, views, urls, user authentication, authorization, models, and deployment. You now have the knowledge to go and build your own websites with Django. The CRUD functionality in our Reviews app is common in many web applications. E.g. you already have all the tools to create a blog, Todo List or shopping cart web applications.

Hopefully, you have enjoyed this book and would like to learn more from me. I would love to get your feedback, understanding what you liked and didn't for us to improve.

Please feel free to email me at support@i-ducate.com to get updated versions of this book.

If you didn't like the book or feel that I should have covered certain additional topics, please email us to let us know. This book can only get better thanks to readers like you.

If you like the book, I would appreciate it if you could leave us a review too. Thank you and all the best for your learning journey in Django development!

About the Author

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Contact Greg at support@i-ducate.com or http://www.greglim.co/

About the Co-Author

Daniel Correa is a researcher and has been a software developer for several years. Daniel has a Ph.D. in Computer Science; currently he is a professor at Universidad EAFIT in Colombia. He is interested in software architectures, frameworks (such as Laravel, Django, Express, Vue, React, Angular, and many more), web development, and clean code.

Daniel is very active on Twitter; he shares tips about software development and reviews software engineering books. Contact Daniel on Twitter at @danielgarax

"Hecho en Medell í n"