1
Installing Python and Django

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 involved in building a website, such as connecting to a database, handling security, enabling user authentication, creating URL routes, displaying content on a page through templates and forms, supporting multiple database backends, and setting up an admin interface.

This reduction in repetitive tasks allows developers to focus on building a web application's functionality rather than reinventing the wheel for standard web application functions.

Django is one of the most popular frameworks available and is used by established companies such as *Instagram*, *Pinterest*, *Mozilla*, and *National Geographic*. It is also easy enough to be used in start-ups and for building personal projects.

Understanding the app we will be building

For our project, we will be building a movie reviews app that will allow users to view and search for movies, as shown in *Figure 1.1*:

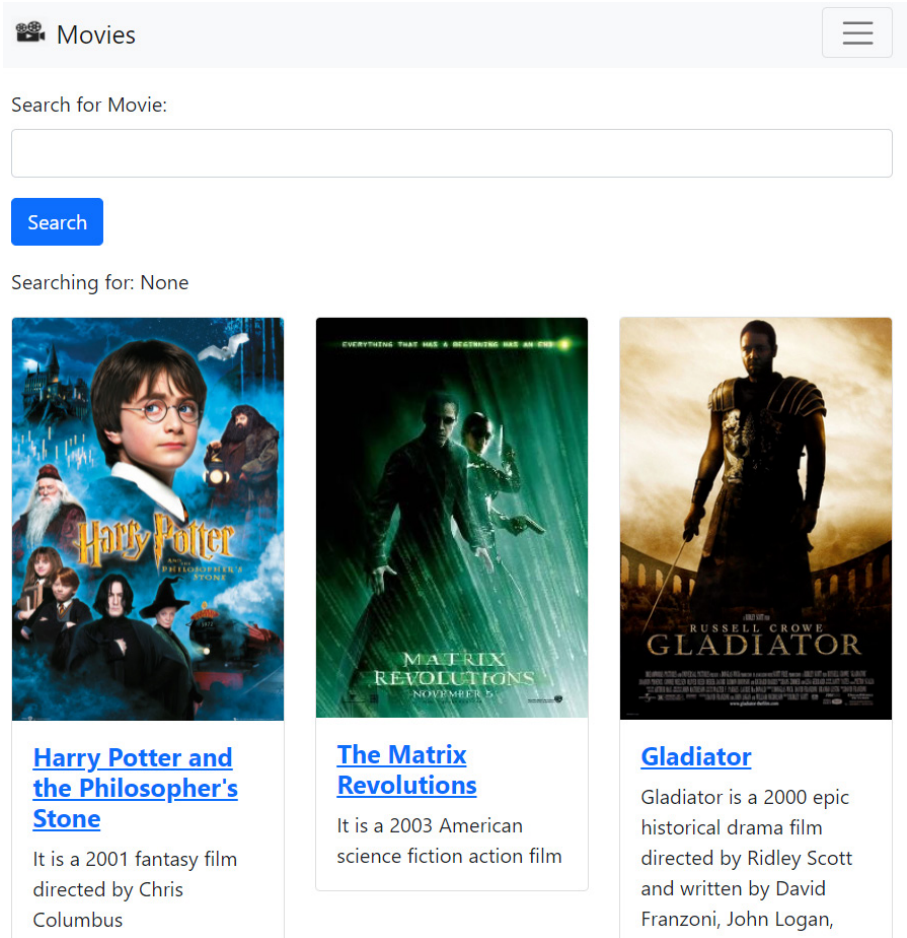


Figure 1.1 – A home page with search functionality

Users will also be able to log in and post reviews of any movies they may have watched, as shown in *Figure 1.2*:

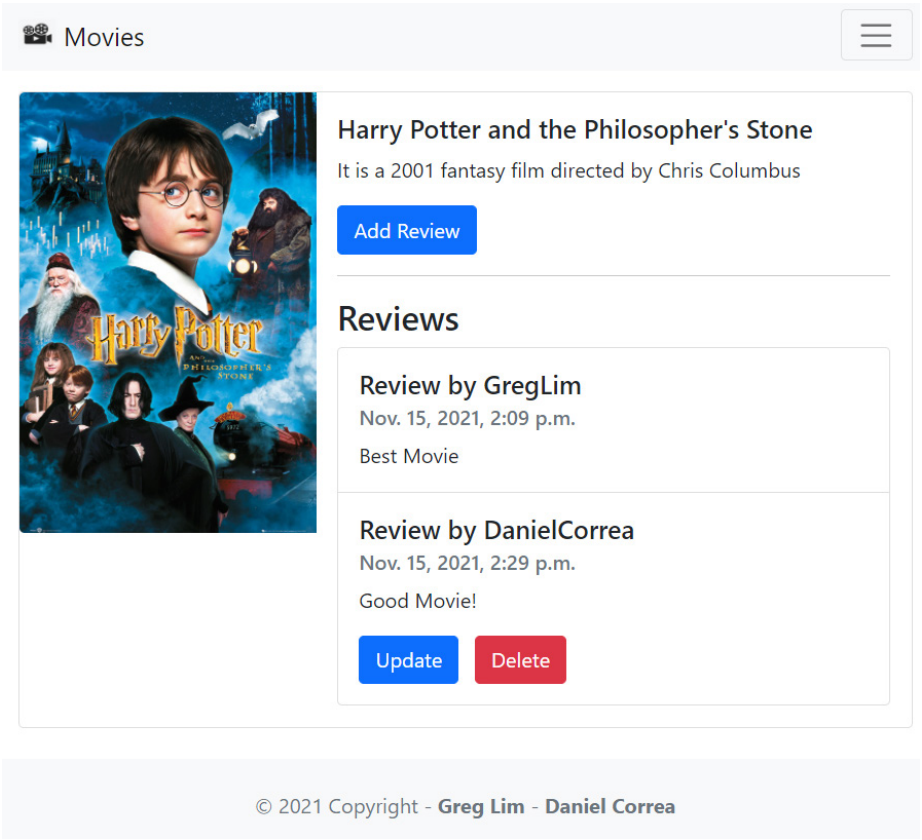


Figure 1.2 – A movie page listing reviews

They will be able to type in and add their review, as shown in *Figure 1.3*:

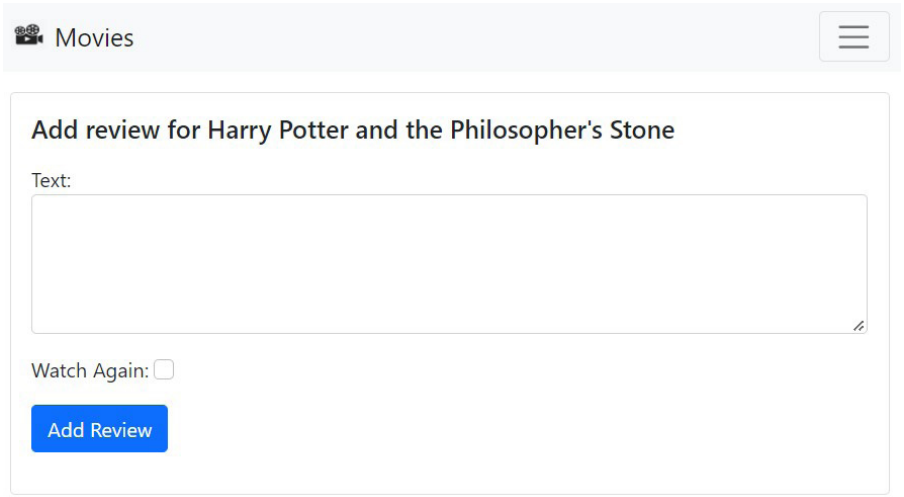


Figure 1.3 – An interface for writing a review

Users can see the list of reviews on a movie's page and post, edit, or delete their own review if they are logged in. They will not be able to edit or delete other users' reviews though. Through building this app, we will learn a lot of concepts, such as forms, user authorization, permissions, foreign keys, and more.

Let's begin by installing Python and Django.

Installing Python

First, let's check whether we have Python installed and, if so, what version we have.

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

We will need to check whether we have at least Python 3.8 in order to use Django 4. To do so, go to your Terminal and run the following commands.

For macOS, run this:

```
python3 --version
```

For Windows, run this:

```
python --version
```

This shows the version of Python you have installed. Make sure that the version is at least *3.8*. If it isn't, get the latest version of Python by going to `https://www.python.org/downloads/` and installing the version for your OS. For Windows, you must select the **Add Python 3.* to PATH** option, as shown in *Figure 1.4*:

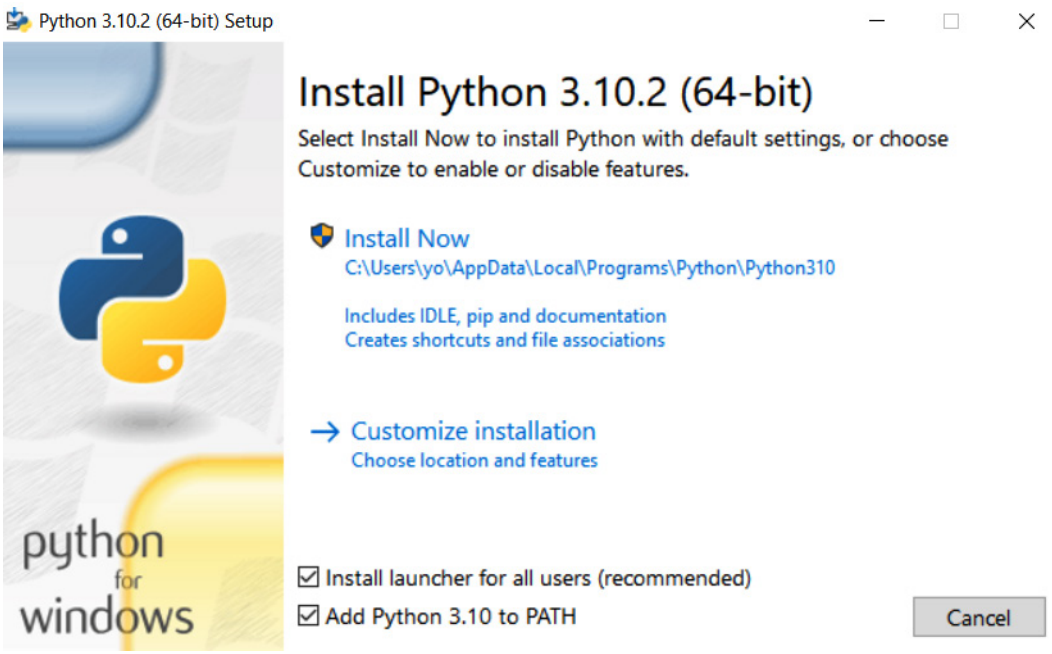


Figure 1.4 – Install Python on Windows

After the installation, run the command again to check the version of Python installed.

The output should reflect the latest version of Python, such as Python 3.10.2 (at the time of 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. pip is automatically installed if you downloaded Python from `https://www.python.org/`.

First, check whether you have pip installed by going to the Terminal and running the following.

For macOS, run this:

```
pip3
```

For Windows, run this:

```
pip
```

If you have pip installed, the output should display a list of pip commands. To install Django, run the following command:

For macOS, run this:

```
pip3 install Django==4.0
```

For Windows, run this:

```
pip install Django==4.0
```

The preceding command will retrieve the latest Django code and install it on your machine. After installation, close and reopen your Terminal.

Ensure you have installed Django by running the following commands.

For macOS, run this:

```
python3 -m django
```

For Windows, run this:

```
python -m django
```

Now, the output will show you all the Django commands you can use, as shown in *Figure 1.5*:

```
MacBook-Air:~ user$ python3 -m django

Type 'python -m django help <subcommand>' fo

Available subcommands:

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    runserver
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver
Note that only Django core commands are list
nfigured (error: Requested setting INSTALLED
ured. You must either define the environment
r call settings.configure() before accessing
```

Figure 1.5 – Django module commands

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

In the Terminal, navigate to a folder on your computer where you want to store your Django project, such as `Desktop`. In that folder, you will run a command like this:

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

In our case, let's say we want to name our project `moviereviews`. We can do so by running the following.

For macOS, run this:

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

For Windows, run this:

```
python -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 the Terminal, `cd` into the created folder:

```
cd moviereviews
```

Then, run the following.

For macOS, run this:

```
python3 manage.py runserver
```

For Windows, run this:

```
python 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, as shown in *Figure 1.6*:

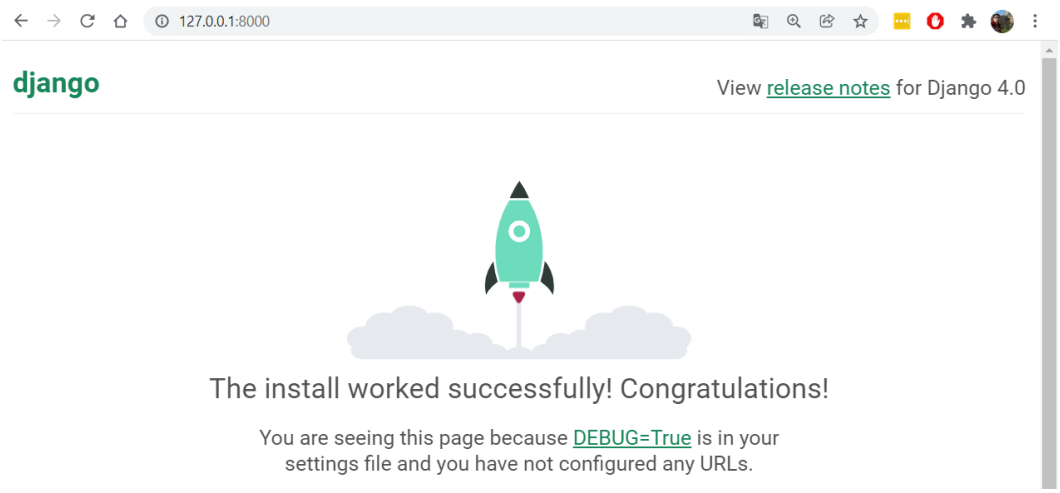


Figure 1.6 – The landing page of the Django project

This means that your local web server is running and serving the landing page. Sometimes, you will need to stop the server in order to run other Python commands. To stop the local server, press *Ctrl* + *C* in the Terminal.

Summary

In this chapter, you learned how to install and use Python, pip, and Django. You also learned how to create a new Django project and run a Django local web server. In the next chapter, we will look inside the project folder that Django has created for us to understand it better.

2
Understanding the Project Structure and Creating Our First App

Understanding the project structure

Let's look at the project files that were created for us in *Chapter 1*, *Installing Python and Django*, in the *Installing Django* section. Open the `moviereviews` project folder in VS Code. You will see the following elements:

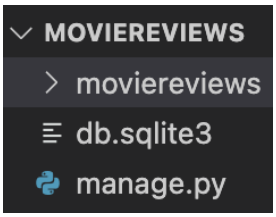


Figure 2.1 – The MOVIEREVIEWS directory structure

Let's learn about each of these elements.

The moviereviews folder

As you can see in *Figure 2.1*, there is a folder with the same name as the folder we opened in VS Code originally – `moviereviews`. To avoid confusion and to distinguish between the two `moviereviews` folders, we will keep the inner `moviereviews` folder as it is and rename the outer folder `moviereviewsproject`.

After the renaming, open the inner `moviereviews` folder. You will see the following elements, as shown in *Figure 2.2*:

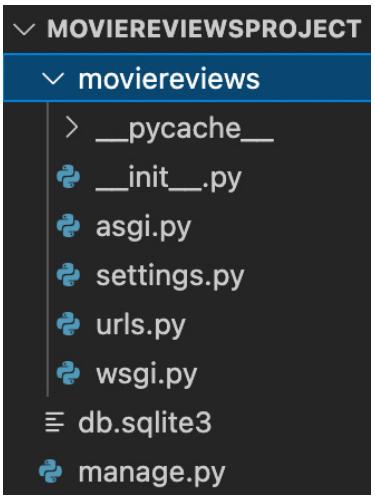


Figure 2.2 – The MOVIEREVIEWSPROJECT directory structure

Let's briefly look at all the elements in the `moviereviews` folder:

- `__pycache__`: This folder 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 can then be readily executed.
- `__init__.py`: This file specifies what to run when Django launches for the first time.
- `asgi.py`: This file allows an optional **Asynchronous Server Gateway Interface** (**ASGI**) to run.
- `settings.py`: The `settings.py` file is an important file that controls our project's settings. It contains several properties:
  - `BASE_DIR`: Determines where on your machine the project is situated.
  - `SECRET_KEY`: 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 mode or not. In debug mode, we get detailed information on errors – for instance, if we try to run `http://localhost:8000/123` in the browser, we will see a **Page not found (404)** error:

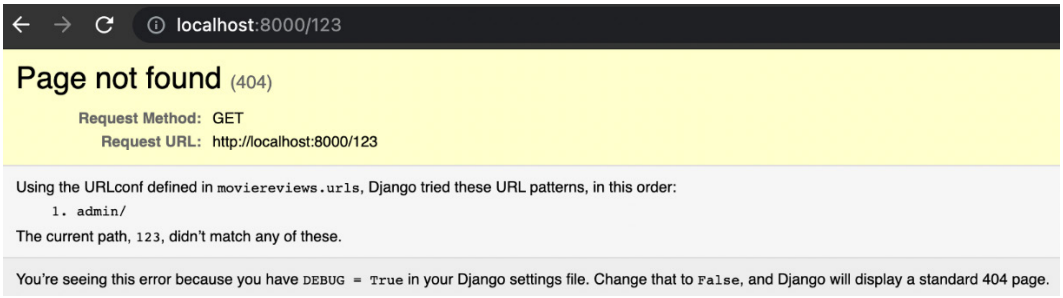


Figure 2.3 – Accessing an invalid application route

> **Note**
>
> It is important to remember the following:
>
> - When deploying our app to production, we should set `DEBUG` to `False`. If `DEBUG` = `False`, we will see a generic 404 page without error details.
> - While developing our project, we should set `DEBUG` = `True` to help us with debugging.

- `INSTALLED_APPS`: Allows us to bring different pieces of code into our project. We will see this in action later.
- `MIDDLEWARE`: Refers to built-in Django functions to process application requests/responses, which include authentication, session, and security.
- `ROOT_URLCONF`: Specifies where our URLs are.
- `TEMPLATES`: Defines the template engine class, the list of directories where the engine should look for template source files, and specific template settings.
- `AUTH_PASSWORD_VALIDATORS`: Allow us to specify the validations that we want on passwords – for example, a minimum length.

There are some other properties in `settings.py`, such as `LANGUAGE_CODE` and `TIME_ZONE`, but we have focused on the more important properties in the preceding list. We will later revisit this file and see how relevant it is in developing our site.

- `urls.py`: This file tells Django which pages to render in response to a browser or URL request. For example, when someone enters the `http://localhost:8000/123` URL, 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.
- `Wsgi.py`: This file stands for the **Web Server Gateway Interface** (**WSGI**) and helps Django serve our web pages. Both files are used when deploying our app. We will revisit them later when we deploy our app.

manage.py

The `manage.py` file seen in *Figure 2.1* and *Figure 2.2* is an element we should not tinker with. The file helps us to perform administrative operations. For example, we earlier ran the following command in *Chapter 1*, *Installing Python and Django*, in the *Running the Django local web server* section:

```
python3 manage.py runserver
```

The purpose of the command was to start the local web server. We will later illustrate more administrative functions, such as one for creating a new app – `python3 manage.py startapp`.

db.sqlite3

The `db.sqlite3` file contains our database. However, we will not discuss this file in this chapter, as we do not need it to create our file. We will do so in *Chapter 5*, *Working with Models*.

Let's next create our first app!

Creating our first app

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, on a movie review site such as *Rotten Tomatoes*, as shown in *Figure 2.4*, we can have an app for listing movies, an app for listing news, an app for payments, an app for user authentication, and so on:

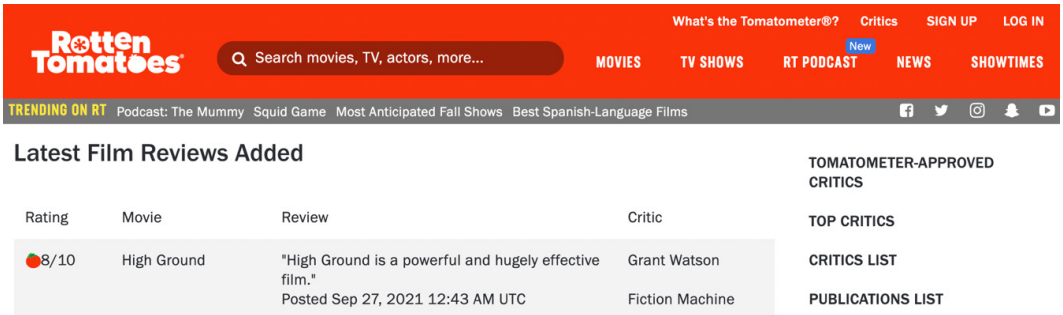


Figure 2.4 – The Rotten Tomatoes website

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 movie 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 by using *Cmd* + *C*. Navigate to the `moviereviewsproject` folder and run a command like the following in the Terminal:

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

In our case, we will add a movie app:

For macOS, run the following command:

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

For Windows, run the following command:

```
python manage.py startapp movie
```

A new folder, `movie`, will be added to the project. As we progress in the book, we will explain the files that are inside the folder.

Although our new app exists in our Django project, Django doesn't recognize it till we explicitly add it. To do so, we need to specify it in `settings.py`. So, go to `/moviereviews/settings.py`, under `INSTALLED_APPS`, and you will see six built-in apps already there.

Add the app name, as highlighted in the following (this should be done whenever a new app is created):

```
…
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:

For macOS, run with the following:

```
python3 manage.py runserver
```

For Windows, run with the following:

```
python manage.py runserver
```

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

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

```
"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 this problem later. But for now, remember that we can have one or more apps inside a project.

Summary

In this chapter, we discussed the Django project structure. We analyzed some of the most important project files and their functionalities. We saw how a web project can be composed of several applications, and we learned how to create a Django app. In the next chapter, we will see how to manage Django routes to provide the project with custom pages. And in upcoming chapters, we will see how the Django architecture model-view-template fits inside the Django project structure.

3
Managing Django URLs

Understanding and defining Django URLs

Remember that `/moviereviews/urls.py` is referenced each time someone types in a URL on our website – for example, `localhost:8000/hello`.

For now, we get an error page when we go to the preceding 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 sees whether the URL matches any defined paths so that the Django server can return an appropriate response.

`urls.py` currently has the following 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 example, if a user enters `http://localhost:8000/admin` into the browser, the URL will match the `admin/` path. The server will then respond with the Django admin site (as shown in *Figure 3.1*), which we will explore later:

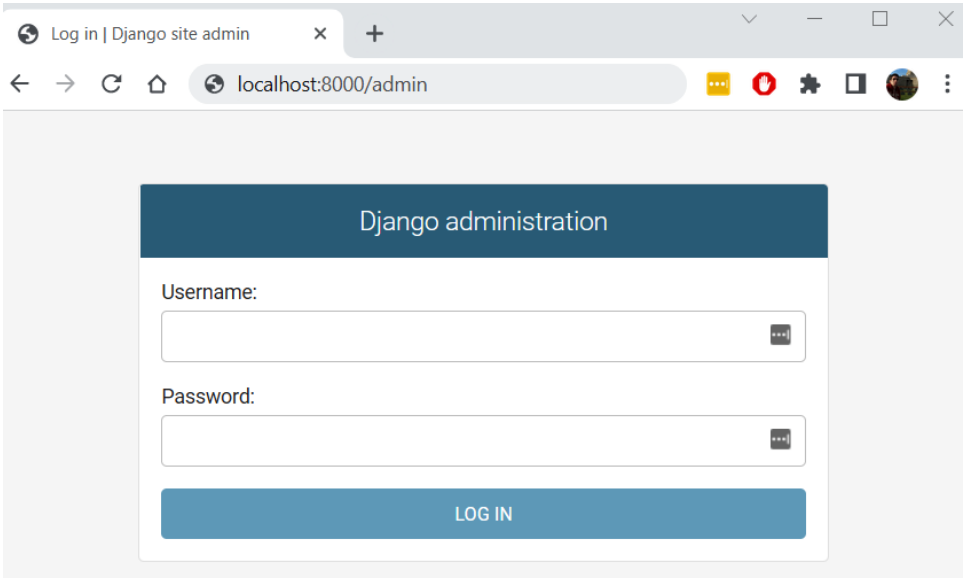


Figure 3.1 – The admin page

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

Creating a custom path for a home page

To illustrate the creation of a custom path, let's create a path for a home page. Add the following code 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),
]
```

Let's explain the preceding code snippet:

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

We added a new path object with the `''` path – that is, it matches the `localhost:8000/` URL 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 that is `/movie/views.py`. Because we have not defined the `home` function in `/movie/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 `home` function that returns an 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 (*Figure 3.2*):

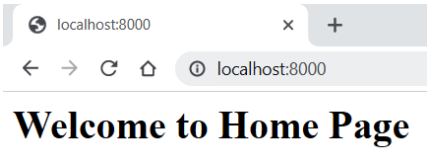


Figure 3.2 – Home page

Congratulations! We have added a new `localhost:8000/` home path 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`. Commonly, About pages contain information about an application and its creators. We will just display a simple message.

Creating a custom path for an about page

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 (*Figure 3.3*):

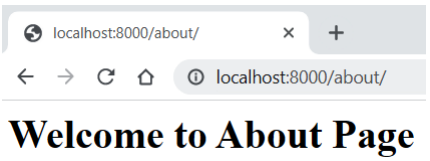


Figure 3.3 – About page

> **Note**
> When we make changes to a file and save it, Django watches the file changes and reloads the server with the changes. Therefore, we don't have to manually restart the server each time there is a code change.

Summary

We now know how to create custom paths and linked them with respective view functions. Note that `urls.py` is located in the project's main folder, `moviereviews`. All requests to the site will go through `urls.py`. Then, specific paths defined in `urls.py` are linked to specific view functions, which are located in the individual app folders. For example, the `/about` path (defined in `urls.py` file) is linked to the `about` function (defined in the `/movie/views.py` file). 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 ideal if we can define the HTML page in a separate file of its own. Let's see how to do so in the next chapter.

4
Generating HTML Pages with Templates

Currently, we have a Django project with URLs connected to Django views. In those views, we defined functions that return HTML code. However, placing HTML code inside Django views is not a good strategy, and affects the project maintainability and evolution. We need to move away from HTML code to their own HTML files.

Understanding Django 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 its own templates folder (*Figure 4.1*).

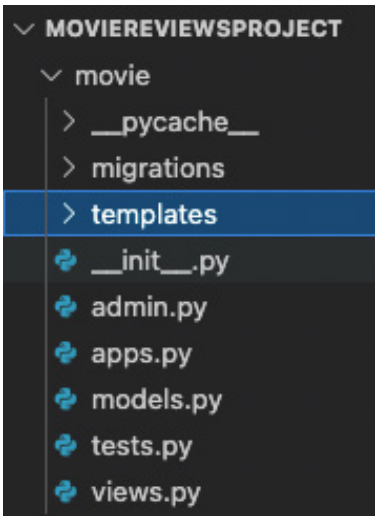


Figure 4.1 – Movie app directory structure

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 refined view for the home page.

Template

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

```
<!DOCTYPE html>
<html>
  <head>
    <title>Movies App</title>
  </head>

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

As you can see, the template simply holds the HTML. It will display two messages with different HTML header tags.

View

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

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

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

Note that 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 business 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 earlier created the URL for our home and about pages 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 the main `''` path (the URL), which links to the `movieViews.home` (view) function that returns the `home.html` (template) code. You will see this pattern of templates, views, and URLs for almost every Django web page. As we repeat this multiple times throughout the book, you will begin to internalize it.

You can see the new home page if you go to `localhost:8000` (*Figure 4.2*):

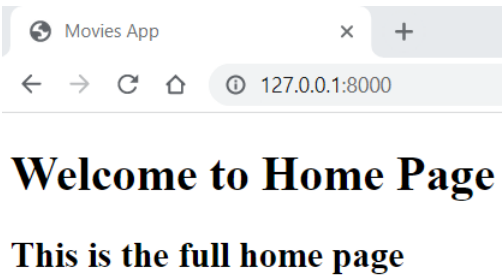


Figure 4.2 – Home page

Now that we have understood how Django templates work, let's look at how we can pass data to the templates.

Passing data into templates

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

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

```
        Lim'})
…
```

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

```
…
  <body>
    <h1>Welcome to Home Page, {{ name }}</h1>
    <h2>This is the full home page</h2>
  </body>
…
```

`{{ name }}` accesses the `'name'` key in the dictionary and thus retrieves the `'Greg Lim'` value. So, if you run the site now and go to the home page, you should see what is shown in *Figure 4.3*:

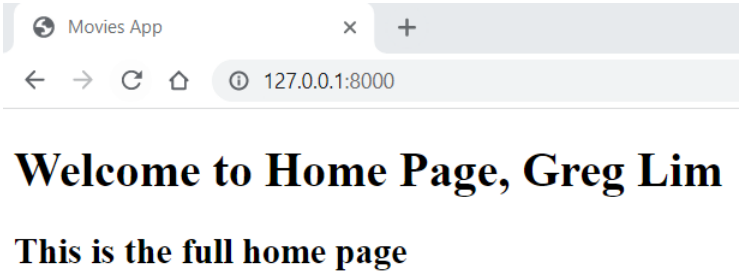


Figure 4.3 – A new home page

In the previous example, we used the variable syntax element from the **Django Template Language** (**DTL**). The DTL provides a set of elements and tags – for example, `{{ ... }}` and `{%. . . %}` – to help render HTML. You can see the full list of built-in elements and tags in the official docs: `https://docs.djangoproject.com/en/4.0/ref/templates/language/`.

We will introduce more DTL elements and tags and their usage as we progress.

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 – for example, a navigation bar, button, alert, list, and card – and simply copy and paste its markup into our template.

Let's look at the steps to do just that:

1. Go to `https://getbootstrap.com/` and go to **Get started** (*Figure 4.4*):

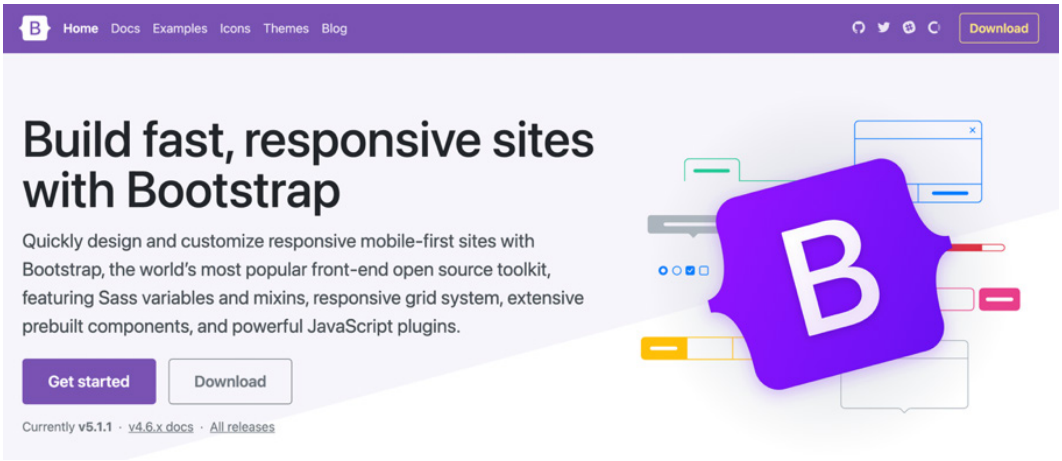


Figure 4.4 – The Bootstrap site

2. Copy the style sheet link (as shown in *Figure 4.5*) inside the `<head>` tag of the `home.html` template to load the Bootstrap CSS:

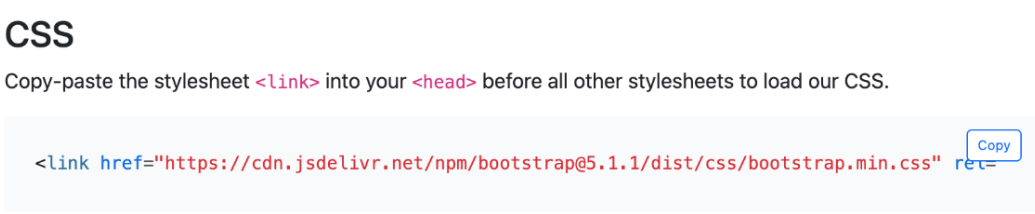


Figure 4.5 – Bootstrap CSS CDN link

`home.html` will look something like this:

```
<!DOCTYPE html>
<html>
  <head>
```

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

3. You can immediately see the styling applied if you go to `localhost:8000` (as shown in *Figure 4.6*):

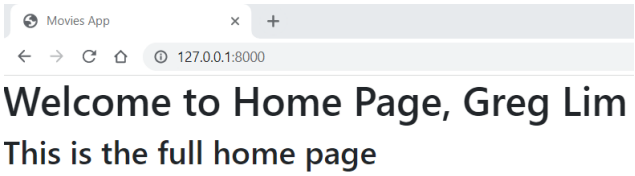


Figure 4.6 – Home page with Bootstrap

4. To improve the padding of the site, let's wrap the `home.html` HTML header tags in a `div` container:

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

The result looks like this:

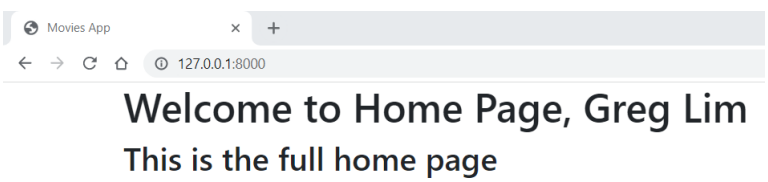


Figure 4.7 – Home page with the container

There are many other Bootstrap components we will use and add to our site. Next, we will use a form from Bootstrap!

Adding a search form

We will add a search form on our home page for users to search for movies. Let's get a Form component from *getbootstrap*. At `https://getbootstrap.com`, under **Docs**, go to **Forms | Overview** (as shown in *Figure 4.8*):

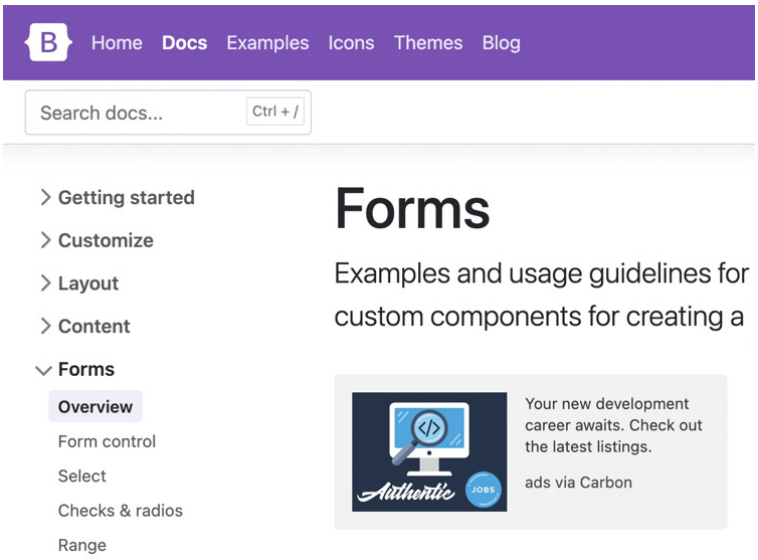


Figure 4.8 – Bootstrap forms

In the **Overview** section, you can copy the markup (as shown in *Figure 4.9*) and paste it inside the `home.html` template file. This is very useful because you will have a skeleton to design HTML forms with Bootstrap.

```
<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 4.9 – Bootstrap forms

Because we don't need much information from the previous form (such as `password` and `checkbox`), we refine the previous HTML code to only contain the **Search for Movie** field and a **Search** button. Add the following in bold to the `home.html` template file:

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

Now, you should have a simple search form (as shown in *Figure 4.10*):

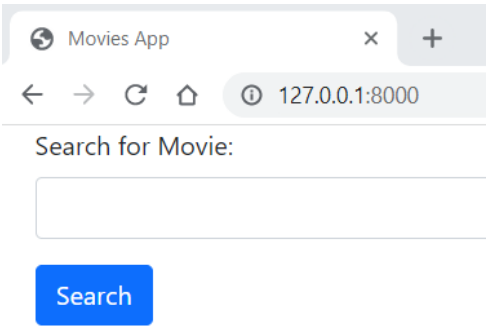


Figure 4.10 – Home page with form

For `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 – that is, `home.html`.
If we want to submit the form to another page – for example, the about page – we will have `<form action="{% url 'about' %}">`. We will illustrate this in the next section. For now, we submit to the same page.

movie/views.py

Now, how do we retrieve the values submitted? Because the form submits to `""`, which `urls.py` routes to `def home` in `movie/views.py`, we can retrieve the values from the request object in `def home` (add the following code in bold):

```
…
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` – that is, `request.GET.get('searchMovie')` – to get the input value. We assign the input value to `searchTerm`.

movie/templates/home.html

We then pass `searchTerm` into `home.html` in the render function with `{'searchTerm':searchTerm}`. Add the following code in bold to `home.html`:

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

When we run our app, enter a value in the search form and hit **Search**:

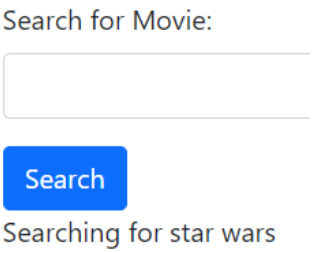


Figure 4.11 – Home page with the form completed

The page reloads and the search term appears after the form (as shown in *Figure 4.11*).

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 that? Let's illustrate by having a **Join our mailing list** form below the search form. Add the following markup to `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 – for example `'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 it's 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 send it to `signup.html` by passing in a key-value pair dictionary, `{'email':email}`.

In `movie/templates/`, create a new file, `signup.html`, with the following markup:

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

```
  <body>
    <div class="container">
      <h2>Added {{ email }} to mailing list</h2>
    </div>
  </body>
</html>
```

When you run your site, add a valid email to the signup form and click **Sign Up** (as shown in *Figure 4.12*):

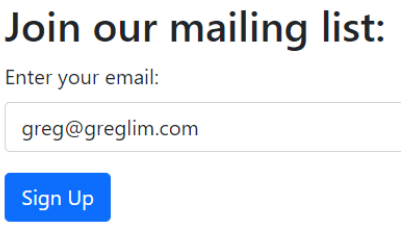


Figure 4.12 – Home page with the signup form filled in

You will be brought to the signup page (`signup.html`) with a response message (as shown in *Figure 4.13*):

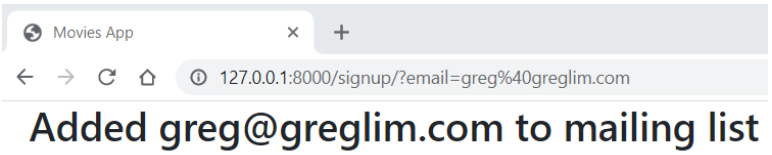


Figure 4.13 – Signup page

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:

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

And in `/moviereviews/urls.py`, add the following 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 home page, `{% url 'home' %}`. We also added `name='about'` to configure a URL route for our about page.

If you go to `signup.html`, you will see the following:

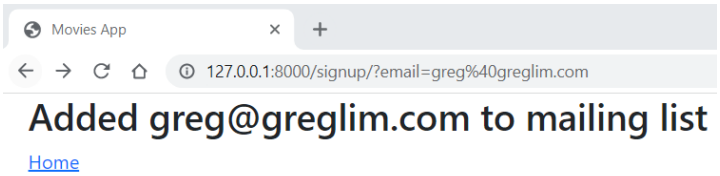


Figure 4.14 – Signup page with a back link

There will be a link to navigate back to home (as shown in *Figure 4.14*).

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`, connects to a Django view, 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 such as models to make our site database-driven.

5
Working with Models

Storing data in a database is a common practice in most web applications. In a Django project, it involves working with Django models. We will create a database model (for example, blog posts and movies) and Django will turn this model into a database table for us. We will also explore a powerful built-in admin interface that provides a visual way of managing all aspects of a Django project, such as managing users and making changes to model data.

Creating our first model

Working with databases in Django involves working with models. A **model** contains the fields and behaviors of the data we want to store. Commonly, each model maps a database table. We can create models such as blog posts, movies, and users, and Django turns these models into a database table for us.

Here are the Django model basics:

- Each model is a class that extends `django.db.models.Model`.
- Each model attribute represents a database column.
- With all of this, Django provides us with a set of useful methods to **create, update, read, and delete** (**CRUD**) model information from a database.

/movie/models.py

In `/movie`, we have the `models.py` file, where we create our models for the movie app. Open that file and 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)
```

Let's look at what happens in this code:

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

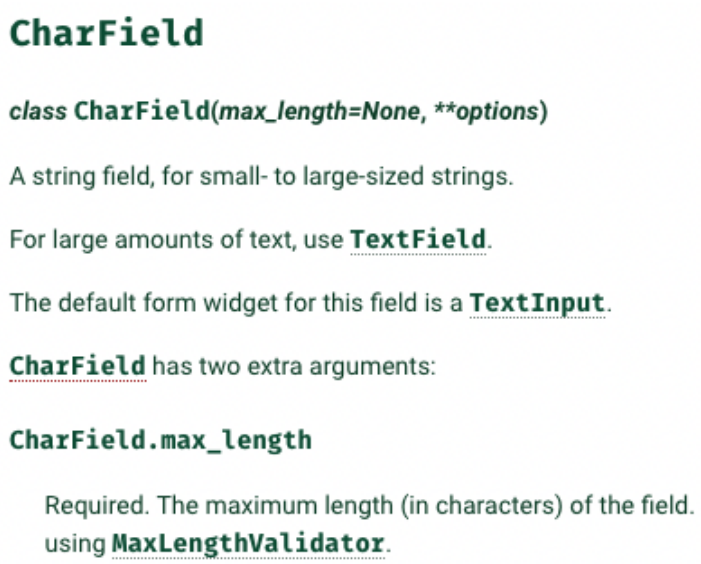
We import the `models` module from `django.db`. This module helps to define and map the characteristics of the model into the database. In our case, we created a `Movie` model to store the `title`, `description`, `image`, and `url` of a movie.

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

`class Movie` inherits from the `Model` class. The `Model` class allows us to interact with the database, create a table, and 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. Note that the properties have types such as `CharField`, `ImageField`, and `URLField`. Django provides many other model fields to support common types such as dates, integers, and emails. 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/4.0/ref/models/fields/`). For example, `CharField` is a string field for small-to-large-sized strings, and the `max_length` argument is required (as shown in *Figure 5.1*):



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 5.1 – CharField documentation

Let's understand the preceding code snippet.

We assign `CharField` to both `title` and `description`. `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` is of `URLField`, a `CharField` for a URL. Because not all movies have URLs, we specify `blank=True` to indicate 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 the following.

For macOS, run this:

```
pip3 install pillow
```

For Windows, run this:

```
pip install pillow
```

Managing migrations

Migrations allow us to generate a database schema based on model code. Once we make changes to our models (such as adding a field and renaming a field), new migrations should be created. In the end, migrations allow us to have a trace of the evolution of our database schema (as a version control system).

Currently, note 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 the following.

For macOS, run this:

```
python3 manage.py migrate
```

For Windows, run this:

```
python manage.py migrate
```

The `migrate` command creates an initial database based on Django's default settings. Note that there is a `db.sqlite3` file in the project root 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. In the previous case, the `migrate` command applied 18 default migrations (as shown in *Figure 5.2*). Those migrations were defined by some default Django apps – `admin`, `auth`, `contenttypes`, and `sessions`. These apps are loaded in the `INSTALLED_APPS` variable in the `moviereviews/settings.py` file. So, the `migrate` command runs the migrations of all the installed apps. Note that `INSTALLED_APPS` also loads the `movie` app. However, no migrations were applied for the `movie` app. This is because we have not generated the migrations for the `movie` app:

```
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 5.2 – Applying default Django migrations

Let's create the migrations for the `movie` app. We will run the `makemigrations` command in the Terminal.

For macOS, run this:

```
python3 manage.py makemigrations
```

For Windows, run this:

```
python manage.py makemigrations
```

This generates the SQL commands for the defined models in all 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. The migrations are stored in an auto-generated folder, `migrations` (as shown in *Figure 5.3*):

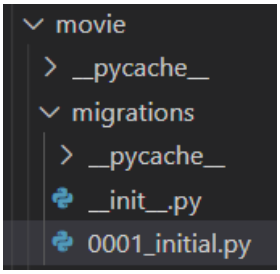


Figure 5.3 – Generated migrations for the movie app

Then, we need to build the actual database with `migrate` (`python3 manage.py migrate`), which executes the SQL commands in the migrations file. Currently, it uses the default SQLite database engine, but you can integrate your own DB system by modifying the `moviereviews/settings.py` file. Now, execute the following in the Terminal.

For macOS, run this:

```
python3 manage.py migrate
```

For Windows, run this:

```
python manage.py migrate
```

As you can see in *Figure 5.4*, we applied the movie migrations:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, movie, sessions
Running migrations:
  Applying movie.0001_initial... OK
```

Figure 5.4 – Applying the movie migrations

In summary, each time you make changes to a model file, you have to run the following.

For macOS, run this:

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

For Windows, run this:

```
python manage.py makemigrations
python 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, which there will be more on in the next section.

Accessing the 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),
    path('about/', movieViews.about),
    path('signup/', movieViews.signup, name='signup'),
]
```

If you go to `localhost:8000/admin`, it brings you to the admin site (as shown in *Figure 5.5*):

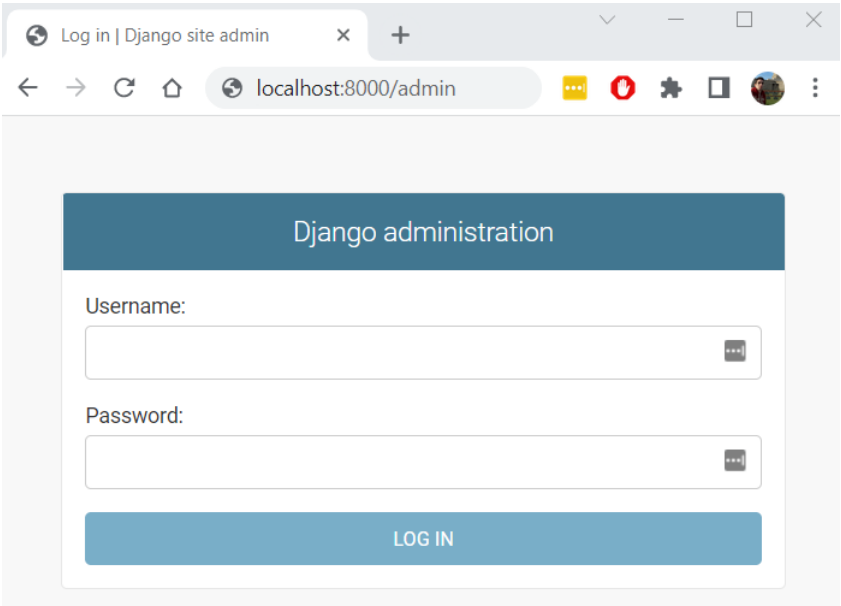


Figure 5.5 – Admin page

Django has a powerful built-in admin interface that provides a visual way of managing all aspects of a Django project – for example, managing users and 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 the following.

For macOS, run this:

```
python3 manage.py createsuperuser
```

For Windows, run this:

```
python 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 that your password is something secure.

If you wish to change your password later, you can run the following commands.

For macOS, run this:

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

For Windows, run this:

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

Then, start the server again, and log into admin with the username you have just created (as shown in *Figure 5.6*):

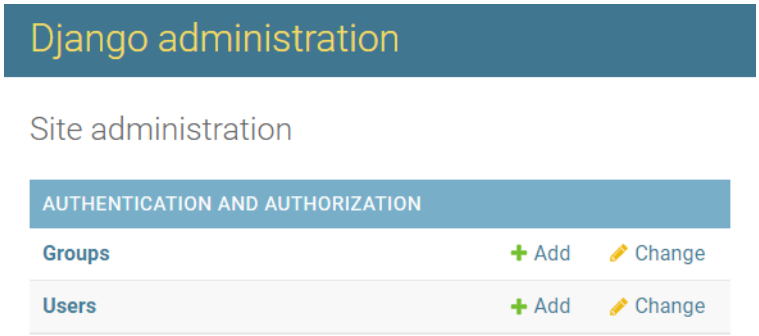


Figure 5.6 – Site administration page

Under **Users**, you will see the user you have just created (as shown in *Figure 5.7*):

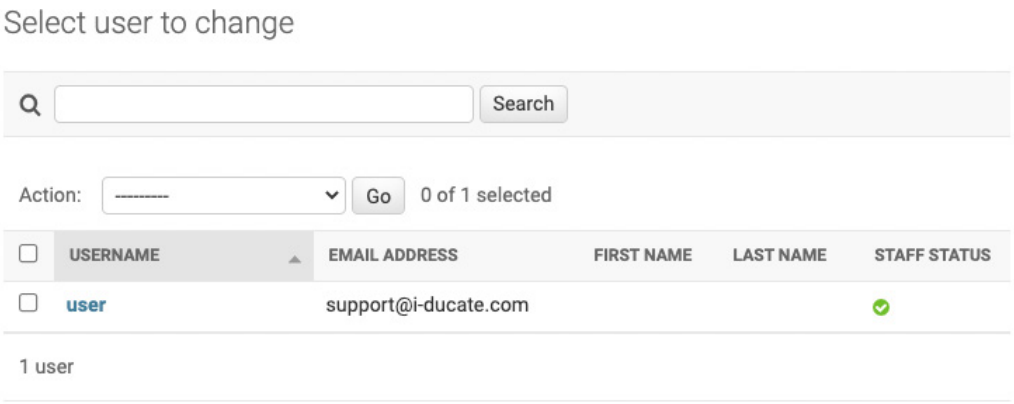


Figure 5.7 – Users admin page

You can add additional user accounts here for your team.

Currently, our movie model doesn't show up in admin. We need to explicitly tell Django what to display in it. 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 the following at the bottom of the file:

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

At the top of the file, add the following:

```
import os
```

Here, `MEDIA_ROOT` is the absolute filesystem path to the directory that will hold user-uploaded files, and we join `BASE_DIR` with `'media'`. Also, `MEDIA_URL` is the URL that handles the media served from `MEDIA_ROOT` (refer to `https://docs.djangoproject.com/en/4.0/ref/settings/` to find out more about the setting properties).

When we add a movie in admin (explained in the *Adding a movie model to admin* section later in this chapter), you will see the image stored inside the `/moviereviews/media/` folder.

Serving the stored images

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

```
…
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. Having configured the images, let's add our movie model to the admin panel.

Adding a movie model to admin

To add the movie model to admin, go back to `/movie/admin.py`, and register our model with the following:

```
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 (as shown in *Figure 5.8*):

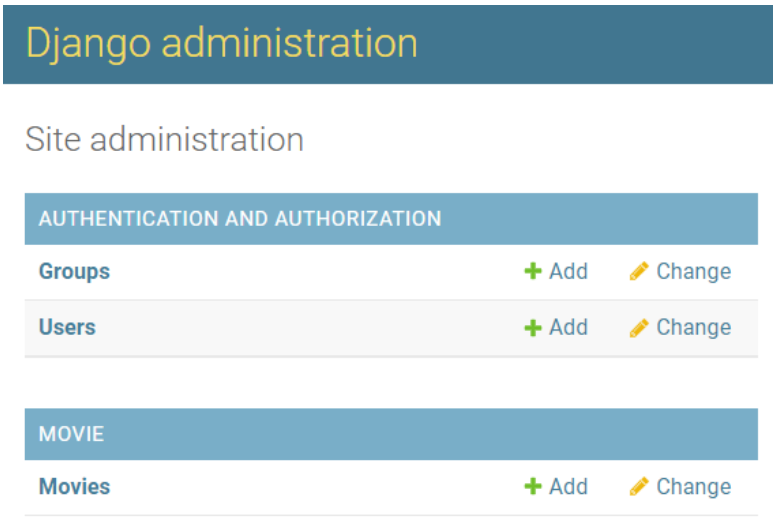


Figure 5.8 – Admin page

Try adding a `movie` object by clicking on **+Add**. You will be brought to the **Add movie** form (as shown in *Figure 5.9*):

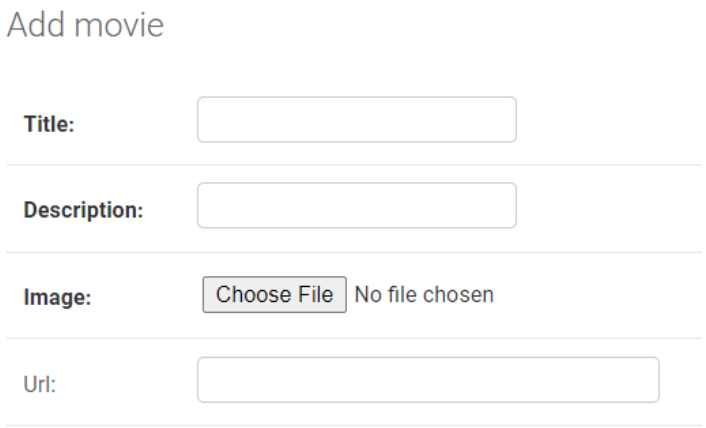


Figure 5.9 – Add movie page

Note that **Url** is not in bold, as we 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**. Your movie object will be saved to the database and reflected in the admin page (as shown in *Figure 5.10*):

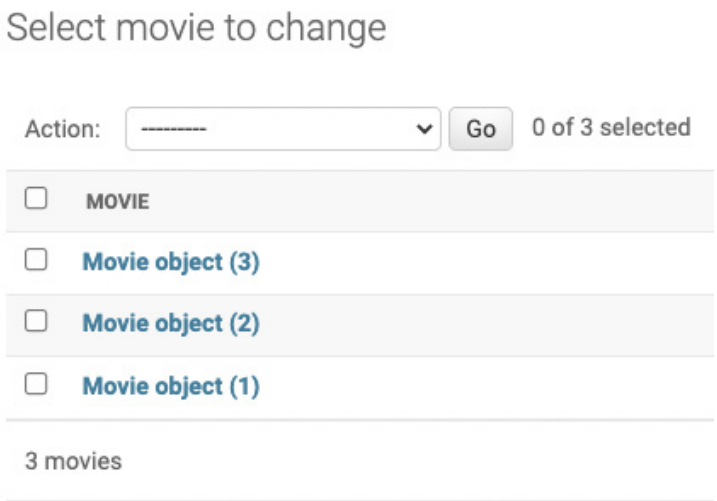


Figure 5.10 – Movies admin page

You can also see the movie image in `/moviereviews/media/movie/images/<image file>.jpg`.

Summary

Models are essential to work with databases in Django. We learned the fundamentals of Django models and created a movie model. We also learned to use the Django admin interface and how to create movies. Now, let's see how we can display these movies on our site in the next chapter.

6
Displaying Objects from Admin

Previously, we learned how to store data in a database. In this chapter, we will collect and display database information. We will display movie information and use some Bootstrap cards to improve the look and feel of our app. Then, we will create a news app with its respective model, view, templates, and URLs. Finally, we will display the news information.

Listing movies

Let's improve our app. We will 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})
…
```

Let's look at what's happening in the code. We first import the `Movie` model:

```
from .models import Movie
```

Then, the previous code grabs all the movie objects from the database (using the `all` method) and assigns them to `movies`:

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

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 will involve a lot more code! But Django provides lots of database functionality to handle these for us. You can find a full list of the available methods here: `https://docs.djangoproject.com/en/4.0/topics/db/queries/`.

/movie/templates/home.html

In `home.html`, we display the objects by adding the following code in bold:

```
…
  <body>
    <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>
      …
```

Let's understand what's happening here:

```
      {% 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 `{% ... %}` template tags. We use `{{ ... }}` to render variables such as a movie's title, description, and image URL, as shown in the following example:

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

Because a movie URL is optional, which means it can be null, we check whether 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 %}
```

We presented a practical use of both the `for` and `if` template tags. You can find a more complete list of template tags here: `https://docs.djangoproject.com/en/4.0/ref/templates/language/#tags`.

When you run your site and go to the home page, you will see the movies you added (in admin) on the page (as shown in *Figure 6.1*):

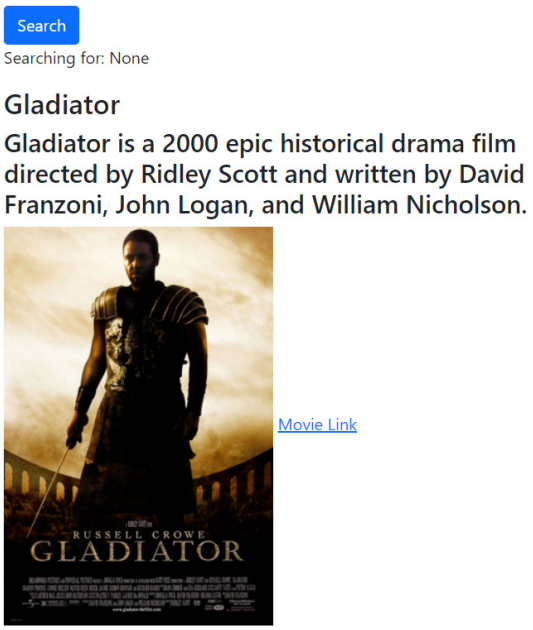


Figure 6.1 – Home page listing movies

If you add another movie in admin, it will be listed on the website when you reload it.

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/`).

Using the card component

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:

```
…
  <body>
    <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>
      …
```

The home page should look something like *Figure 6.2*:

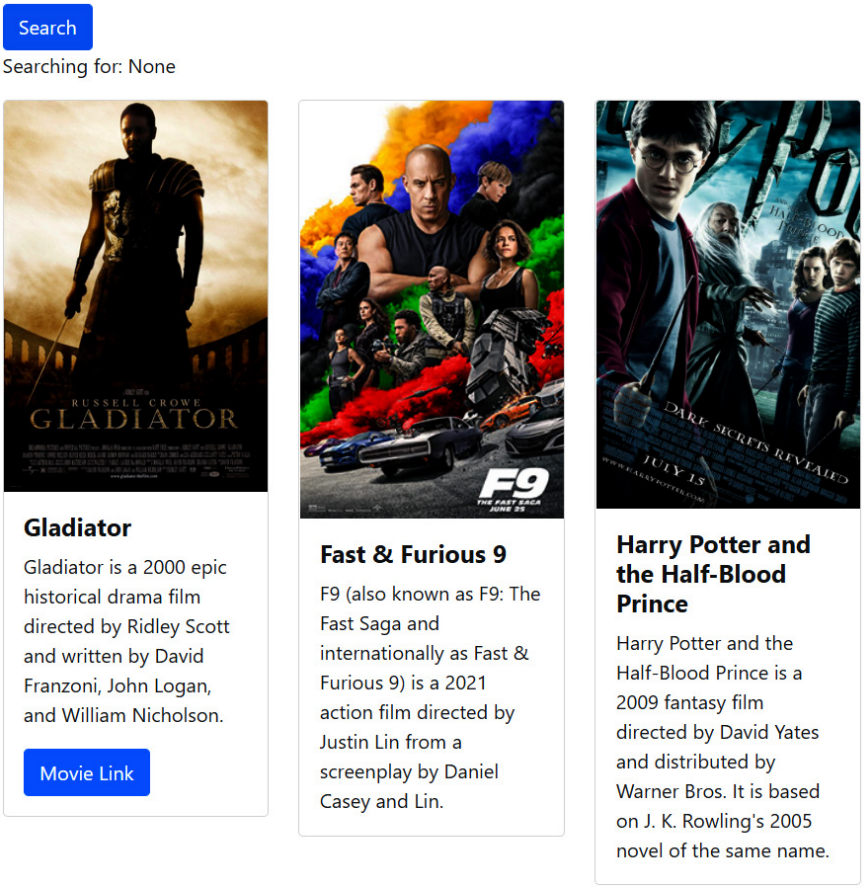


Figure 6.2 – Home page listing movies with the card component

We are currently listing all movies in our database. In the next section, let's display only the movies that fit the user-entered search term.

Implementing a search

Implement `def home` in `/movie/views.py` with the following (remove the old `def home` function and paste the next one):

```
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})
```

Let's see what's happening in the code. We retrieve the search term entered (if any) from the `searchMovie` input:

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

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:

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

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

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

Now, when you run your app and enter a search term, the site displays only the movies that fit the search term.

Up to this point, we have covered a lot of material. Let's now crystallize and recap 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.

Adding a news app

Do you remember how to add an app, add a model, and then display 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 the following in the Terminal:

- macOS:

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

- Windows:

```
python manage.py startapp news
```

A `news` folder will be added to the project.

/moviereviews/settings.py

Each time we add an app, we have to tell Django about it by adding it to `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`. Note 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 would increase, and it would 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 its own `urls.py`.

First, in `/moviereviews/urls.py`, add the following in bold:

```
…
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 the `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 preceding path forwards a request – for example, `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.

News model

Let's first create the `News` model:

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

   ```
   from django.db import models

   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:

2. For macOS, make migrations with the following:

   ```
   python3 manage.py makemigrations
   python3 manage.py migrat
   ```

   For Windows, make migrations with the following:

   ```
   python manage.py makemigrations
   python manage.py migr3te
   ```

3. Next, register the news model by going to `/news/admin.py` and adding the following in bold:

```
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 (as shown in *Figure 6.3*):

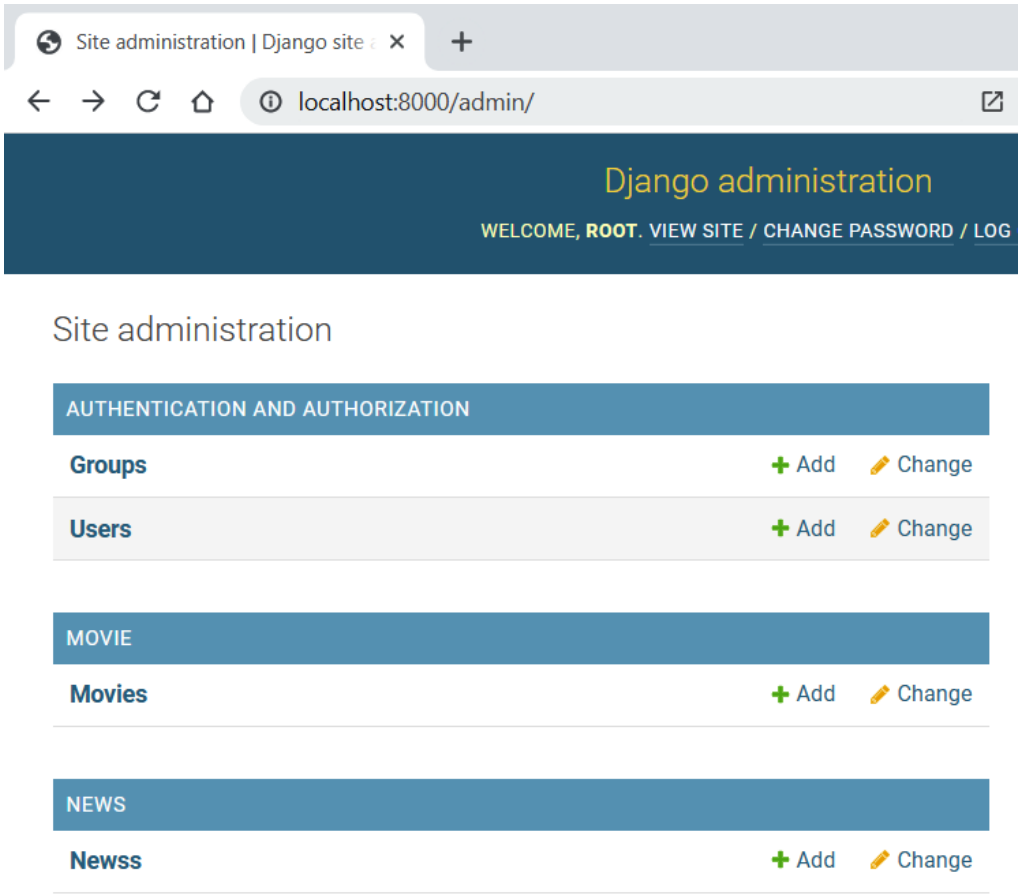


Figure 6.3 – Admin page with news

Listing news

Let's go ahead and display the news articles in `news.html`:

1. In `/news/views.py`, add the following 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 them to `news.html`.

2. In `/news/templates/news.html`, display the news objects with the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Movies App</title>
    <link href="https://cdn.jsdelivr.net/npm/
      bootstrap@5.1.1/dist/css/bootstrap.min.css"
      rel="stylesheet" crossorigin="anonymous">
  </head>
  <body>
    <div class="container">
      {% for news in newss %}
      <h2>{{ news.headline }}</h2>
      <h5>{{ news.date }}</h5>
      <p>{{ news.body }}</p>
      {% endfor %}
    </div>
  </body>
</html>
```

3. Now, try adding some news objects in admin. When you visit `http://localhost:8000/news/`, you should see them displayed (as shown in *Figure 6.4*):

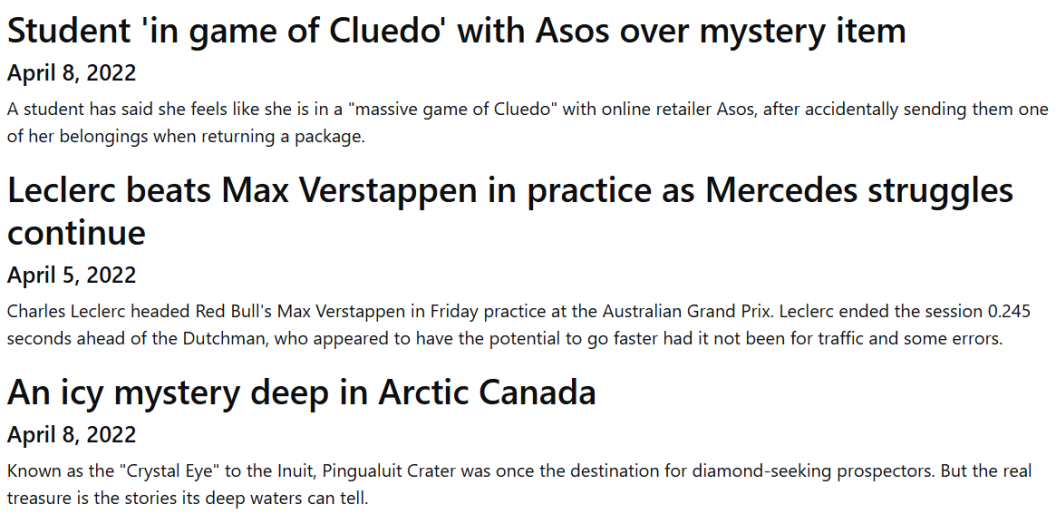


Figure 6.4 – News page

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`:

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

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 (as shown in *Figure 6.5*):

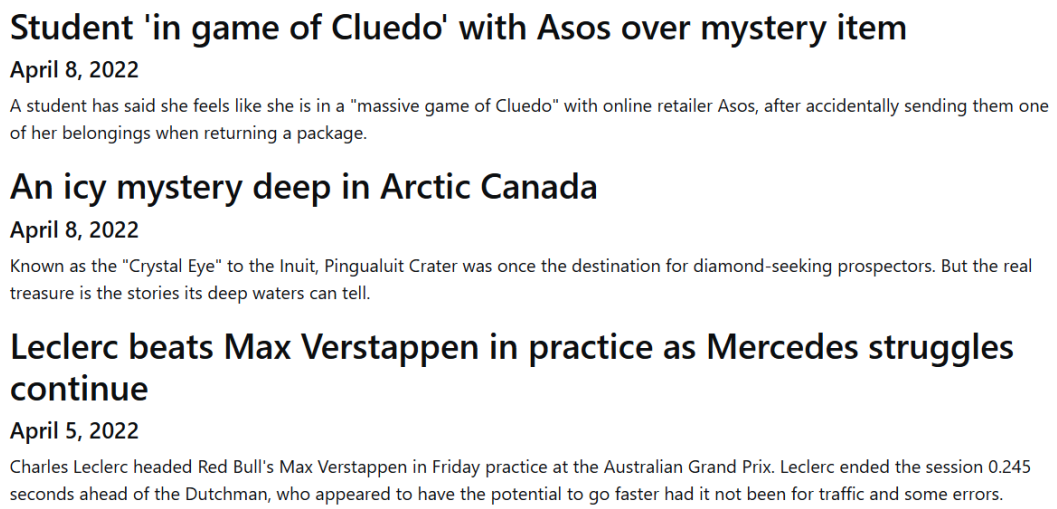


Figure 6.5 – News page with news ordered by date

We should also improve the look of the news site with the Bootstrap Horizontal Card component (`https://getbootstrap.com/docs/5.1/components/card/#horizontal`). In `/news/templates/news.html`, replace the `for` loop markup and make the following changes in bold:

```
…
  <body>
    <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>
    </body>
</html>
```

You should get something like *Figure 6.6*:

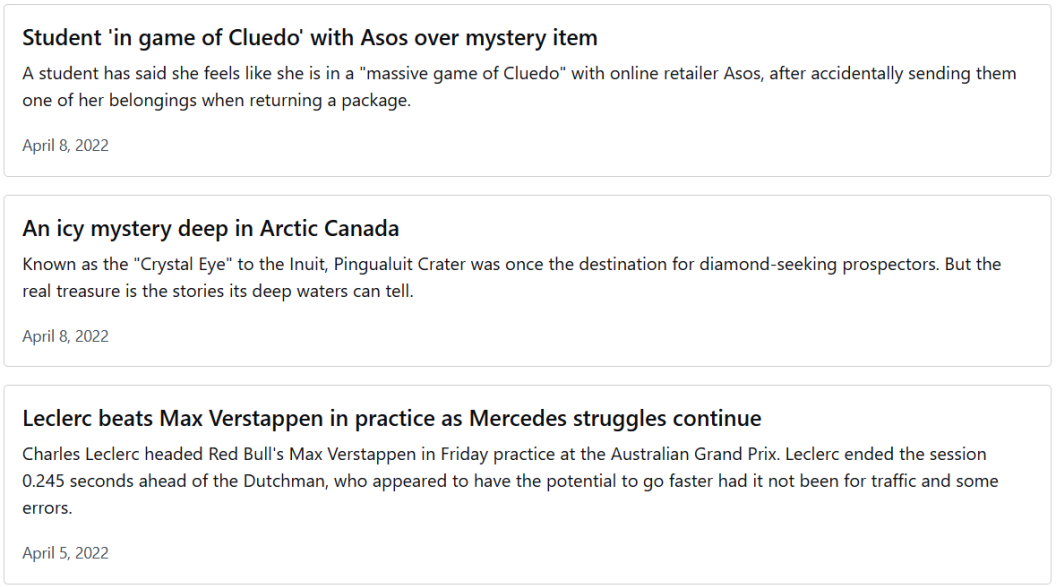


Figure 6.6 – News page with the horizontal card component

Summary

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

7
Understanding the Database

Displaying object information in admin

Currently, when we look at our model objects in admin, it is hard to identify individual objects (as shown in *Figure 7.3*) – for example, **News object (1)** and **News object (2)**:

Figure 7.3 – News admin page

For better readability in admin, we can customize what is displayed there – for example, if we want to display the headline for each news object instead. In `/news/models.py`, add the following 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. Note how readability is improved (as shown in *Figure 7.4*)!

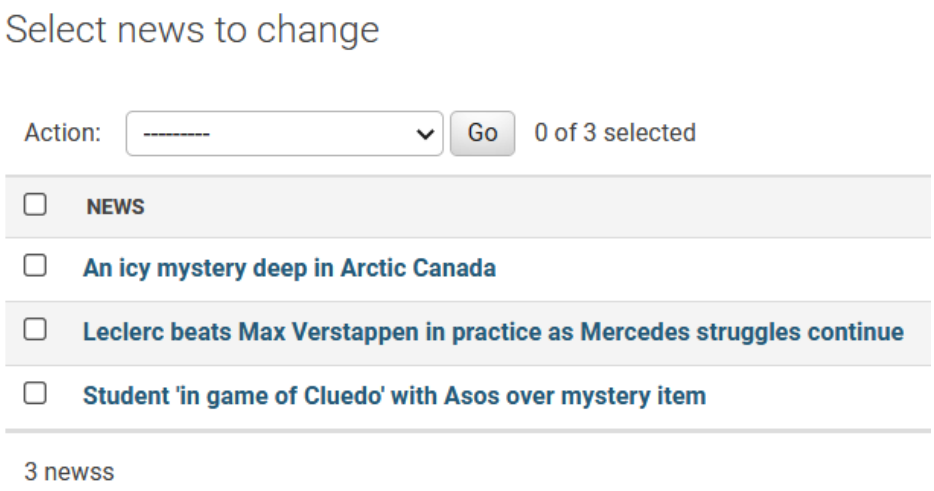


Figure 7.4 – News admin page with headlines

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

8
Extending Base Templates

This chapter presents the concept of base templates and how they can be used to reduce duplicated template code. We will create a base template with a navbar and a footer and also add links to the different app pages.

Creating a base template

We currently have our movies page, mailing list signup page, and news page. However, 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 will begin with `movie/templates/home.html`:

1. We will use as a base the markup of the Navbar component from **getbootstrap** (`https://getbootstrap.com/docs/5.1/components/navbar/`). 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. Besides that, we will include a `meta viewport` tag that detects a user device and scales an application, depending on that device.
2. In `movie/templates/home.html`, make the following changes in bold:

```
<!DOCTYPE html>
<html>
  <head>
    …
    <script src="https://cdn.jsdelivr.net/npm/
      bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js"
      crossorigin="anonymous">
    </script>
    <meta name="viewport" content="width=device-width,
       initial-scale=1" />
  </head>

  <body>
    <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">
      …
```

We have added the navbar to `home.html` (as shown in *Figure 8.1*):

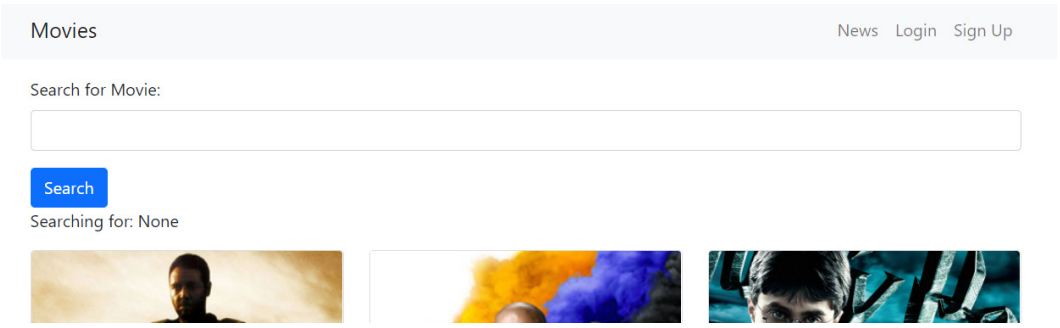


Figure 8.1 – Home page with navbar

We also included two links (**Login** and **Sign Up**), which will be used later.

If you reduce the browser window width, your navbar will respond accordingly (this is automatically provided by the Bootstrap elements we used, as shown in *Figure 8.2*):

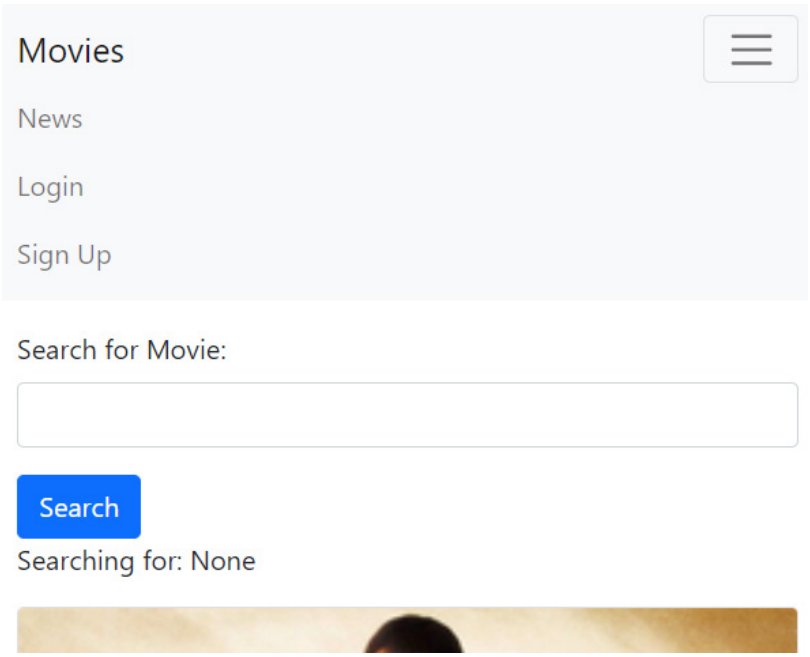


Figure 8.2 – Home page with navbar (reduced window width)

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 would have to add the link on 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`. We will move the common HTML elements (such as the header and navbar) to the `base.html` file.

> **Note**
>
> We can name `base.html` anything, but using `base.html` is a common convention for base templates.

3. Fill `moviereviews/templates/base.html` with the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Movies App</title>
    <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>
    <meta name="viewport" content="width=device-width,
      initial-scale=1" />
  </head>

  <body>
    <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>
  </body>
</html>
```

Let's understand what is happening here.

`base.html`, as its name suggests, serves as the base for all pages. Thus, we include the header navbar in it. We will later include a footer section.

We then allocate a block where content can be slotted in from other child pages – for example, `home.html` and `news.html`:

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

This will become clear as we proceed.

4. Finally, we need to register the `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')],
        'APP_DIRS': True,
        …
```

5. In `movie/templates/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 `<div class="container">` tag, since it is loaded in the `base.html` template. The entire `movie/templates/home.html` will look something like this:

```
{% extends 'base.html' %}
{% block content %}
  <form action="">
    <div class="mb-3">
      <label class="form-label">Search for Movie:
      </label>
      <input type="text" name="searchMovie"
        class="form-control" />
    </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">
    {% 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 %}
```

Let's understand what is happening here.

`base.html`, as its name suggests, serves as the base for all pages. Thus, we include the header navbar in it. We will later include a footer section.

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

With the `extends` tag, we extend from `base.html` by taking all the markup inside the `block content` tag in `home.html` and putting it into `base.html` (inside the `block content` section).

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

6. Now, let's apply the preceding to `news.html` as well. Replace all existing content of `/news/template/news.html` with the following:

```
{% 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 (as shown in *Figure 8.3*):

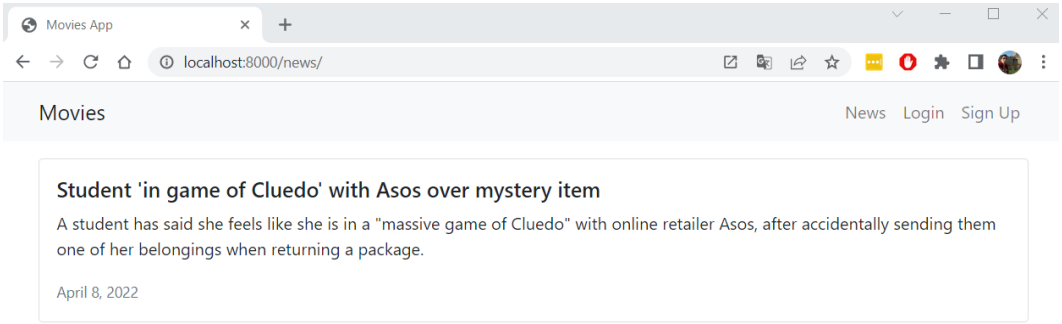


Figure 8.3 – News page extending the base.html template

Now, whenever you want to make a change to the navbar, you just have to do it once in `base.html`, and it will apply to all the pages!

Now, let's configure the header links.

Making the links work

The links in the navbar don't currently work. To enable them, in `moviereviews/templates/base.html`, add the following code in bold:

```
…
  <body>
    <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 earlier defined the `'home'` path in `/moviereviews/urls.py`:

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

We also defined the `'news'` path in `/news/urls.py`:

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

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

Finally, let's include a footer section in our `base.html` template.

Adding a footer section

Let's add a footer section inside our `base.html`. In `moviereviews/templates/base.html`, we will 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">
        © 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>
  </body>
</html>
```

The footer section displays a grey div in which we place the book authors' names, with links to their respective Twitter accounts. If you run the app now, it should give you something like *Figure 8.4*:

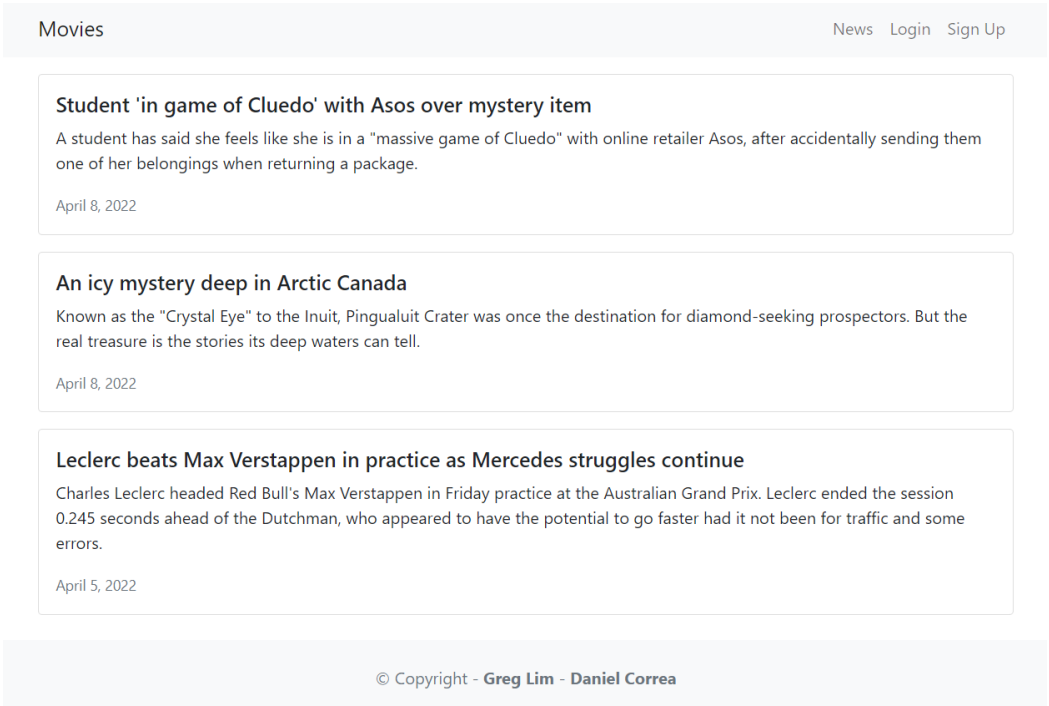


Figure 8.4 – News page

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

Serving static files

Let's suppose we want to display an image icon for our site (as shown in *Figure 8.5*):

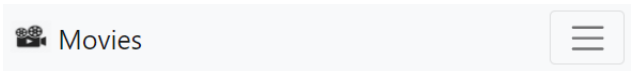


Figure 8.5 – A header with a static image icon

These are examples of fixed images on the site. These fixed images are static files. They are different from media files that users upload to the site, such as movie images.

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

```
…
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.0/howto/static-files/

STATIC_URL = 'static/'
…
```

However, we will go through it here:

1. In `/moviereviews`, create a folder, `static`. In it, create an `images` folder to contain the fixed images used on our site.
2. Bring in an image file (for example, `movie.png`) that you want to display on your site into the `images` folder. In `moviereviews/templates/base.html`, replace the `Movies` link with the following:

```
…
  <body>
    <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>
        …
```

We added an image to the navbar. To add static files to our template, we add the following line to the top of `base.html`, and because other templates inherit from `base.html`, we only have to add this once:

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

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

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

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 in the application settings.

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

```
…

STATICFILES_DIRS = [
    BASE_DIR / 'moviereviews/static/',
]
```

Now, the brand image will be displayed in the navbar (as shown in *Figure 8.6*):

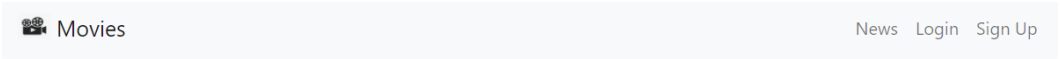


Figure 8.6 – The header with an icon

As well as images, you can link to other static file types – for example, PDF, CSS, and JavaScript files.

Summary

In this chapter, we learned how to create base templates that reduce duplicated code. We improved our application interface with the inclusion of a navbar and footer, and we learned how to manage static files. In the next chapter, we'll see how to navigate to a movie's detail page from the home page.

9
Creating a 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. We will create the movie URL, view, and template.

Creating a movie page

Let's look at the steps:

1. First, we will 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 the movie app's `urls.py`. We put all the paths related to the movie app in its own `urls.py` – that is, `/movie/urls.py`.

2. So, let's create the `/movie/urls.py` file with the following code:

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

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

   The path shown in the previous code block 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 – for example, `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.

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

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

```
from django.shortcuts import get_object_or_404
from .models import Movie

...

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_id`. If there is a match, `get_object_or_404`, as its name suggests, returns us the object or the `not found (404)` object.

4. 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 this:

```
{% 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.

5. If you visit a movie's detail URL, such as `localhost:8000/movie/1`, it will render the movie's details in its own page (as shown in *Figure 9.1*):

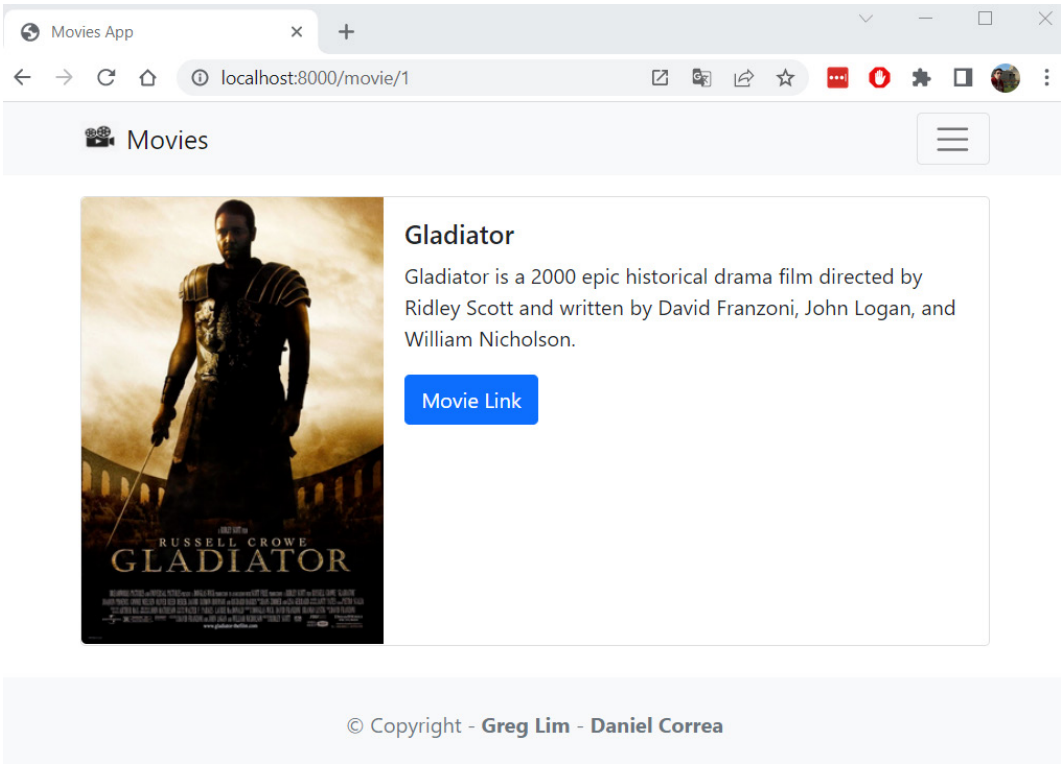


Figure 9.1 – Movie page

Later, in *Chapter 11*, *Letting Users Post, List, Update and Delete Movie Reviews*, we will add a Reviews section to the details page.

Implementing links to individual movie pages

Now that we have a movie's detail page, we will next implement the movie links to navigate to those detail pages from `/movie/templates/home.html`:

1. We simply wrap the movie title in `<a href...>`, as shown here in bold:

```
…
{% 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.

2. When you run your app now and go to the home page, the title will appear as a link to the detail page (as shown in *Figure 9.2*):

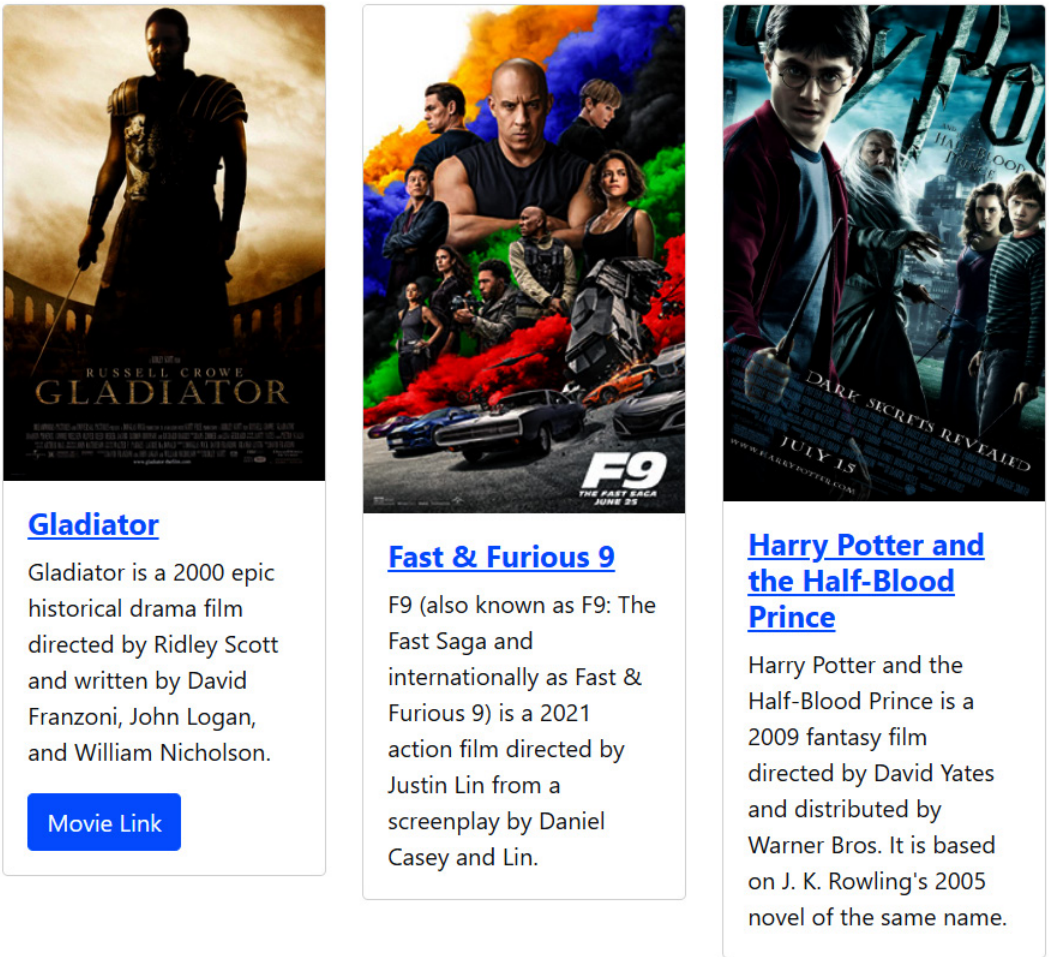


Figure 9.2 – Home page with movie links

Summary

We have learned a lot so far! We have learned about databases, models, the admin interface, static files, media files, extending base templates, URLs, routing URLs, and much more. In this chapter, we added a movie detail page to our application. In the next chapter, we will learn how to allow a user to sign up and log in.

10
Implementing User Signup and Login

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

Creating a signup form

On 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:

1. Since a signup account doesn't belong to the movie or news app, let's create a dedicated app called `accounts` for it. In the Terminal, run the following:

   - For macOS, run this:

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

   - For Windows, run this:

   ```
   python manage.py startapp accounts
   ```

2. 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',
]
…
```

3. We will 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')),
]
…
```

4. We will put all the paths related to the `accounts` app (for example, signup, login, and logout) in its own `urls.py` – that is, `/accounts/urls.py` (create this file with the following code):

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

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

5. Next, create `def signupaccount` in `/accounts/views.py` with the following code in bold:

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

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

6. We will import `UserCreationForm`, which Django provides to easily create a signup form to register new users. Django forms are a vast topic in itself, and we will see how powerful they can be. We pass in the form to `signupaccount.html`.
7. Next, create `/accounts/templates/signupaccount.html` and simply fill in the following:

```
{{ form }}
```

8. Run your app and go to `localhost:8000/accounts/signupaccount`. You will see a form with three fields – username, password1, and password2 (as shown in *Figure 10.1*):

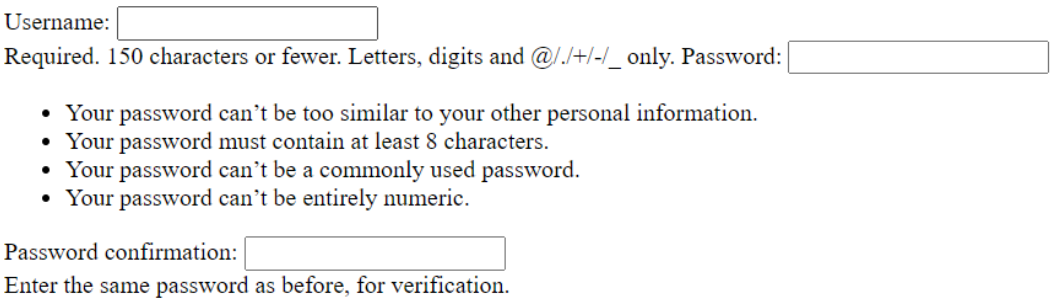


Figure 10.1 – Signup page

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

9. 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 entire `/accounts/templates/signupaccount.html` 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">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 %}
```

The form doesn't have an action, which means that it submits to the same page. We will show how this action is executed later in the *Creating a user* section. Note also that the form method is `POST` – that is, when the form submits, it sends a `POST` request, since we are sending data to the server. This is different from the signup 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 – for example, `http://localhost:8000/signup/?email=greg%40greglim.com`. Because the 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, such as `UPDATE`, `DELETE`, and `PUT`, but they 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.

10. When you run your site, the form will now look something like *Figure 10.2*:

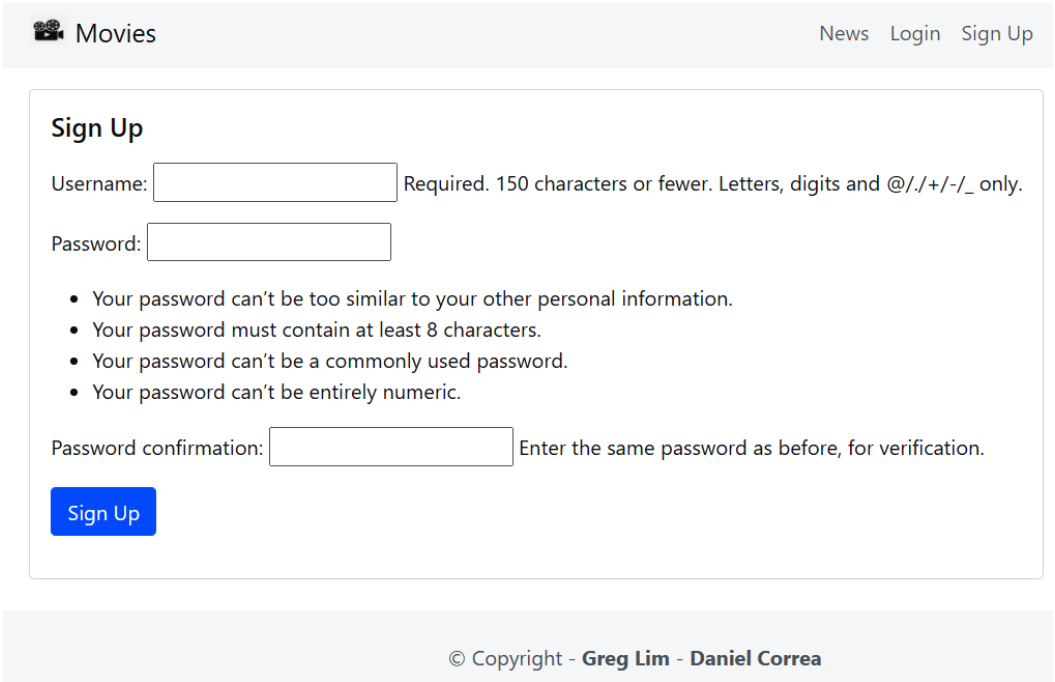


Figure 10.2 – Signup page

Now, let's see how to handle the request and create a user in admin when the user submits the signup form.

Creating a user

When the user submits the signup form, we will have to handle the request and create a user in admin. Use the following steps to do so:

1. Add the following in bold 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')
```

In `def signupaccount`, we first check whether the request received is a `GET` or `POST` request:

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

If it is a `GET` request, it means that it's a user navigating to the signup form via the `localhost:8000/accounts/signupaccount` URL, 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, so we move to the `else` block to create a new user.

In the `else` block, we ensure that the password entered in `password1` and `password2` are the same before going on to create the user:

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

What are `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.

We then retrieve the data entered into the `username` field (`request.POST['username']`) and the `password1` field (`password=request.POST['password1']`):

```
user = User.objects.create_user(
  request.POST['username'],
  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 set up for us. If you recall, in admin, we have **Users** (as shown in *Figure 10.3*), which contains the superuser account created for us when we ran `python3 manage.py createsuperuser`:

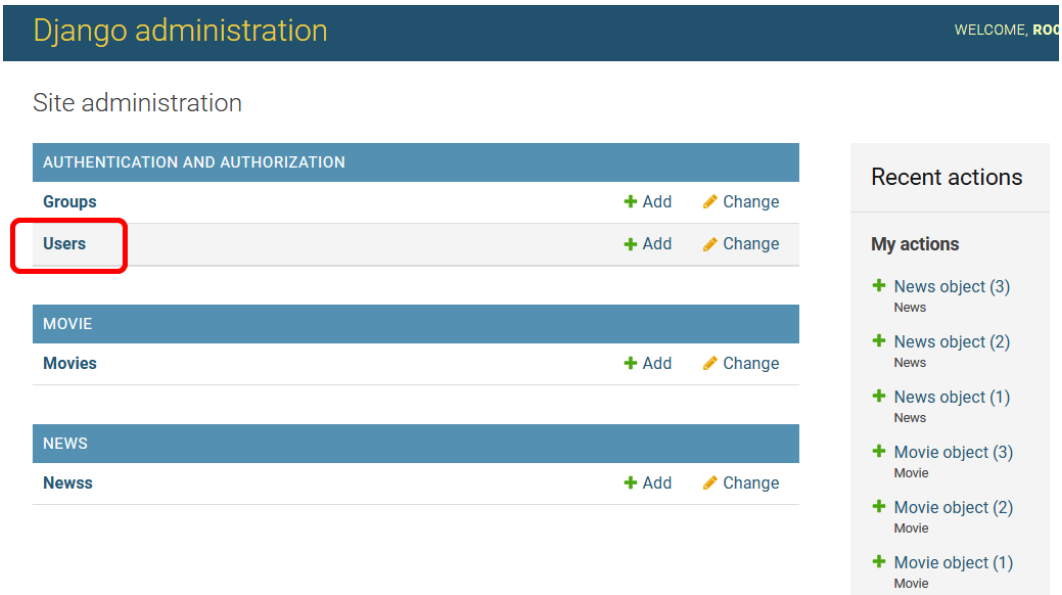


Figure 10.3 – Admin page

`user.save()` actually inserts the new user into the database:

```
user.save()
```

The newly added user will show up in `Users` in admin. After creating the user, we then log in with the new user:

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

This means that, after someone signs up, we automatically log them in and redirect them to the home page.

2. Run your app now and go to `localhost:8000/accounts/signupaccount`. Create a user, and you will see the new user added to admin.

Now that we've built our signup form, let's see how we can resolve any errors that may arise while a user attempts to sign up.

Handling user creation errors

Let's improve the signup form to handle some errors.

Checking whether passwords do not match

What happens if `password1` doesn't match `password2`? To handle such an error, we add the following `else` block in bold in `/accounts/views.py`:

```
…

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')
        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`.

But we also need to render the error message into `signupaccount.html`. In `/accounts/templates/signupaccount.html`, add the following code in bold:

```
{% 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">
        …
```

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

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

We render the error message in a bootstrap `alert` component:

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

This is so that a user can better notice the error and take the necessary corrective action (as shown in *Figure 10.4*):

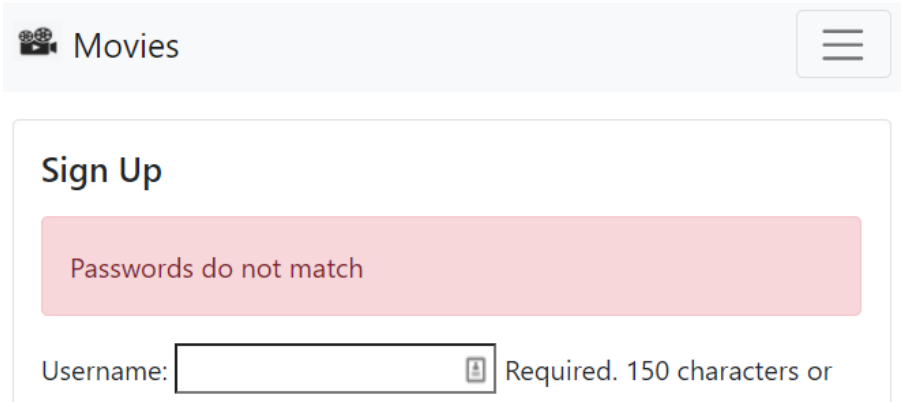


Figure 10.4 – Signup page with the passwords error

Checking if a username already exists

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

There can be other kinds of errors that are identified only from a database – for example, if a 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 here (include this code in `/accounts/views.py`):

```
…
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:
            return render(request, 'signupaccount.html',
             {'form':UserCreationForm,
              'error':'Passwords do not match'})
```

We will import `IntegrityError`, and using `try-except`, we will catch `IntegrityError` when it is thrown (in the case of a username already existing, as shown in *Figure 10.5*):

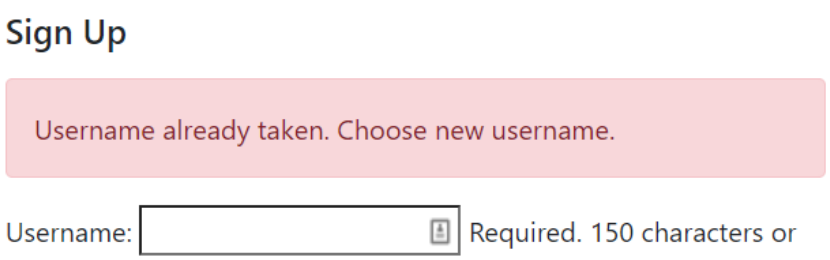


Figure 10.5 – Signup page with a duplicated username error

Now, let's learn how to customize the `UserCreationForm`.

Showing whether 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 (as shown in *Figure 10.7*):

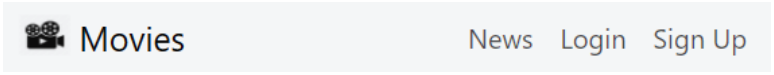


Figure 10.7 – The navbar for the logged in user

For logged-in users, we should be hiding these buttons and showing 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 `moviereviews/templates/base.html`, add the following code 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>
      …
```

Note that we have a `user` object, which Django automatically provides (via the installed auth app) and passes in for us:

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

The user object contains the `username`, `password`, `email`, `first_name`, and `last_name` properties. Additionally, we can check whether a user is logged in with `{% if user.is_authenticated %}`.

If a user is authenticated, we render a **Logout** button with the corresponding username, `{{ user.username }}`. Otherwise, it means that the user is not logged in, and we show the **Login** and **Sign Up** buttons.

Implementing the logout functionality

Create the logout path in `/accounts/urls.py` with the following in bold:

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

In `/accounts/views.py`, implement the `logoutaccount` function with the following in bold:

```
…
from django.contrib.auth.models import User
from django.contrib.auth import login, logout
from django.shortcuts import redirect
from django.db import IntegrityError


…

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

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

We then need to have `<a href>` in the logout button call the logout path. We also have the signup button call the `signupaccount` path. So, in `moviereviews/templates/base.html`, add the following 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 (as shown in *Figure 10.8*):

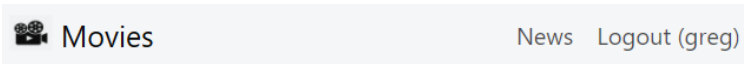


Figure 10.8 – The navbar with the Logout button implemented

Implementing the login functionality

Having implemented signup and logout, let's now implement login:

1. 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'),
]
```

2. In `/accounts/views.py`, implement `loginaccount` with the following in bold:

```
…
...
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')
```

`loginaccount` will be similar to `signupaccount`:

```
def loginaccount(request):
    if request.method == 'GET':
```

```
        return render(request, 'loginaccount.html',
                      {'form':AuthenticationForm})
```

We first handle the case where the request is a `GET` request – that is, the user clicks on `Login` on the navbar – and we render `loginaccount.html`. We then pass in `AuthenticationForm`. Much like `UserCreationForm` for signup, Django provides `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` (the user submits the login form and sends a `POST` request), we proceed to authenticate the user with the values they entered in the username and password fields.

If the user returned from `authenticate` is `None` – that is, we are unable to find an existing user with the supplied username/password – we return the user to `loginaccount.html` with the `'username and password do not match'` error.

Otherwise, it means the authentication is successful, and we log in the user and redirect them to the home page:

/accounts/templates/loginaccount.html

3. Create the new file, `accounts/templates/loginaccount.html`, 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 %}
```

4. Finally, in `moviereviews/templates/base.html`, we set `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 signup links. When a user logs in, the navbar will show only the **Logout** button.

Summary

In this chapter, we implemented a complete authentication system. Now, users can sign up, log in, and log out. We also learned how to take advantage of the Django User model, `AuthenticationForm`, and `UserCreationForm`, and we even learned how to extend some of those classes.

In the next chapter, we will implement a complete movie review system.

11

Letting Users Create, Read, Update, and Delete Movie Reviews

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:

1. 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
```

The `text` field stores the review text:

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

For the review `date`, we specify `auto_now_add=True`:

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

This means that, when someone creates this object, the current datetime will be automatically filled in.

For the `user` and `movie` fields, we are using `ForeignKey`, which allows for a many-to-one relationship:

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

This means that `user` can create multiple reviews. Similarly, `movie` 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 `ForeignKey`, we must also specify an `on_delete` option. This means that when you remove a user or movie, for instance, its associated reviews will be deleted as well. Note that this does not apply in the other direction – that is, when you remove a review, the associated movie and user still remain:

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

Lastly, we have a Boolean property, `watchAgain`, for users to indicate whether they will watch the movie again.

2. 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)
```

3. In the terminal, make the migration for the new model and apply the changes to the sqlite3 database:
   - For macOS, use these:

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

   - For Windows, use these:

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

Now, let's see how to let users post reviews on movies from the site.

Creating a review

We have seen how to create model objects from the admin – for example, creating a `movie` object. But how do we allow users to create their own objects, such as letting them 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:

1. 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

2. In `/movie/views.py`, add `def createreview`:

```
from django.shortcuts import render
from django.http import HttpResponse
from django.shortcuts import get_object_or_404, redirect
from .models import Movie, Review
from django.contrib.auth.decorators import login_required
```

```
    …
```

```
def createreview(request, movie_id):
    movie = get_object_or_404(Movie,pk=movie_id)
    if request.method == 'GET':
        return render(request, 'createreview.html', {'movie': movie})
    else:
        try:
            myreview = request.POST.get('myreview')
            newReview = Review()
            newReview.user = request.user
            newReview.movie = movie
            newReview.text = myreview
            newReview.save()
            return redirect('detail', newReview.movie.id)
        except ValueError:
            return render(request, 'createreview.html', {'error':'bad data
passed in'})
```

3. In `/movie/templates`, create a new file, `createreview.html`, with the following 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">
```

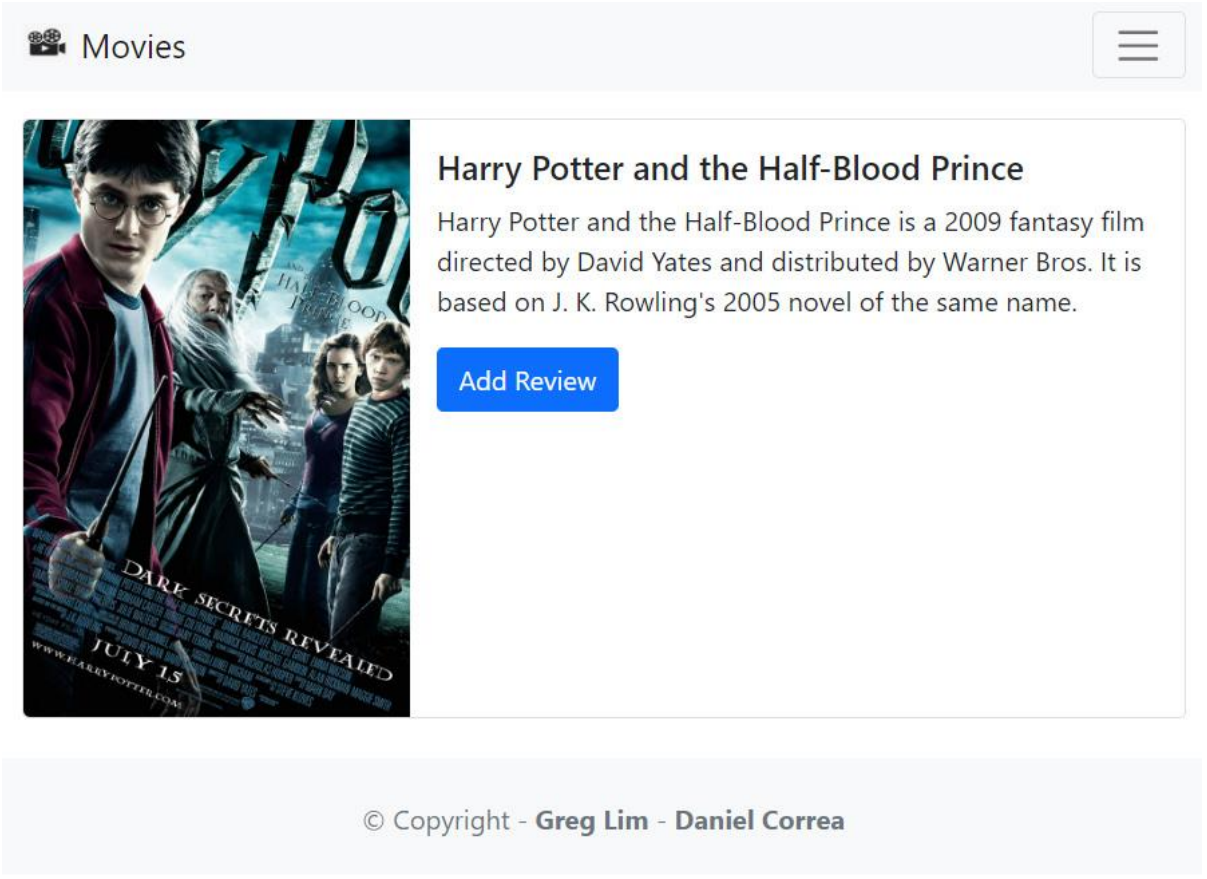
```
                        {% csrf_token %}
                        <div class="form-control">
                            <textarea name="myreview" rows="4" cols="50">
                            </textarea>
                        </div>
                        <button type="submit" class="btn btn-primary">
                            Add Review
                        </button>
                    </form>
                </p>
            </div>
        </div>
    </div>
</div>
{% endblock content %}
```

5. Finally, we render an `Add Review` button on the movie details page (`/movie/templates/detail.html`) with the following codes in bold:

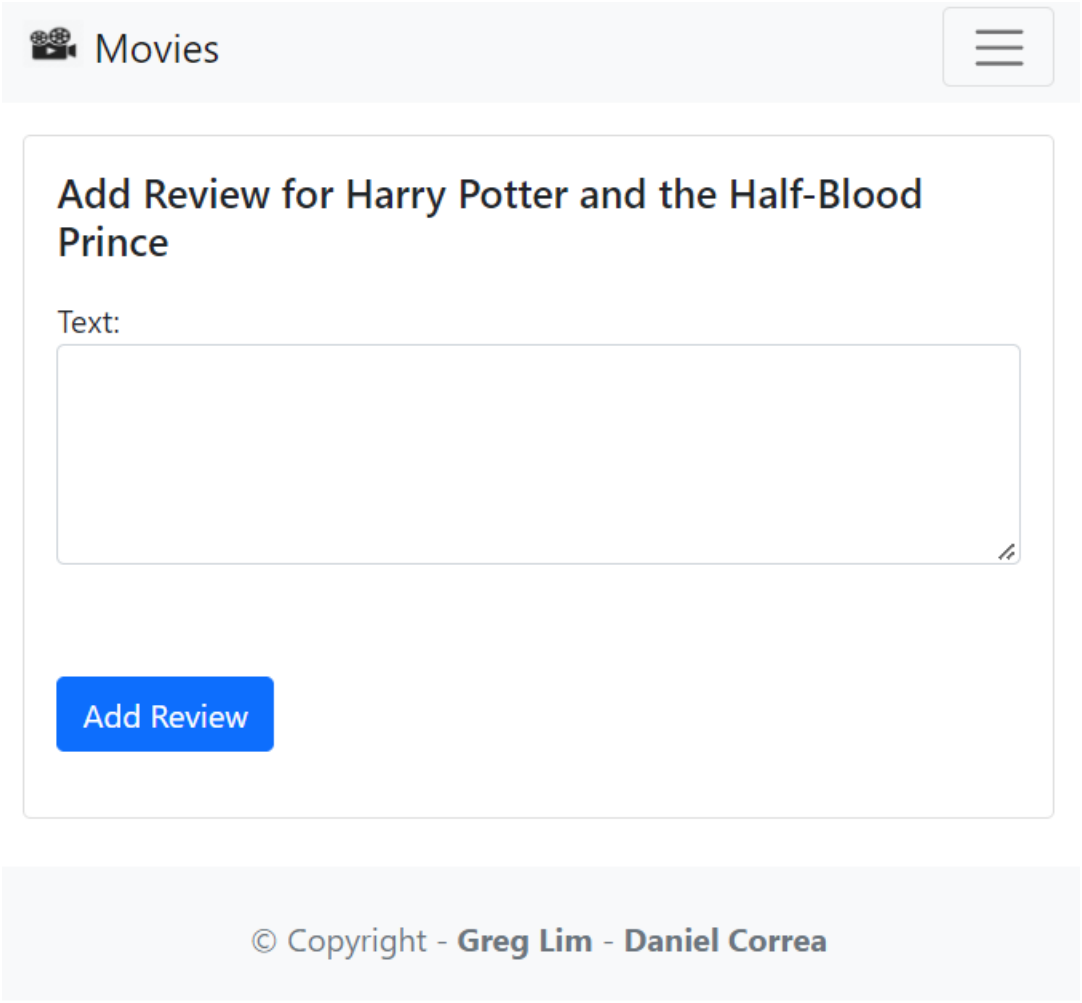
```
...
      <div class="card-body">
        <h5 class="card-title">{{ movie.title }}</h5>
        <p class="card-text">{{ movie.description
          }}</p>
        <p class="card-text">
          {% if movie.url %}
            <a href="{{ movie.url }}"
              class="btn btn-primary">
              Movie Link
```

```
            </a>
          {% endif %}
          {% if user.is_authenticated %}
          <a href="{% url 'createreview' movie.id %}"
            class="btn btn-primary">
            Add Review
          </a>
          {% endif %}
        </p>
      </div>
...
```

6. Log in, go to a movie, and click **Add Review** (*Figure 11.1*):



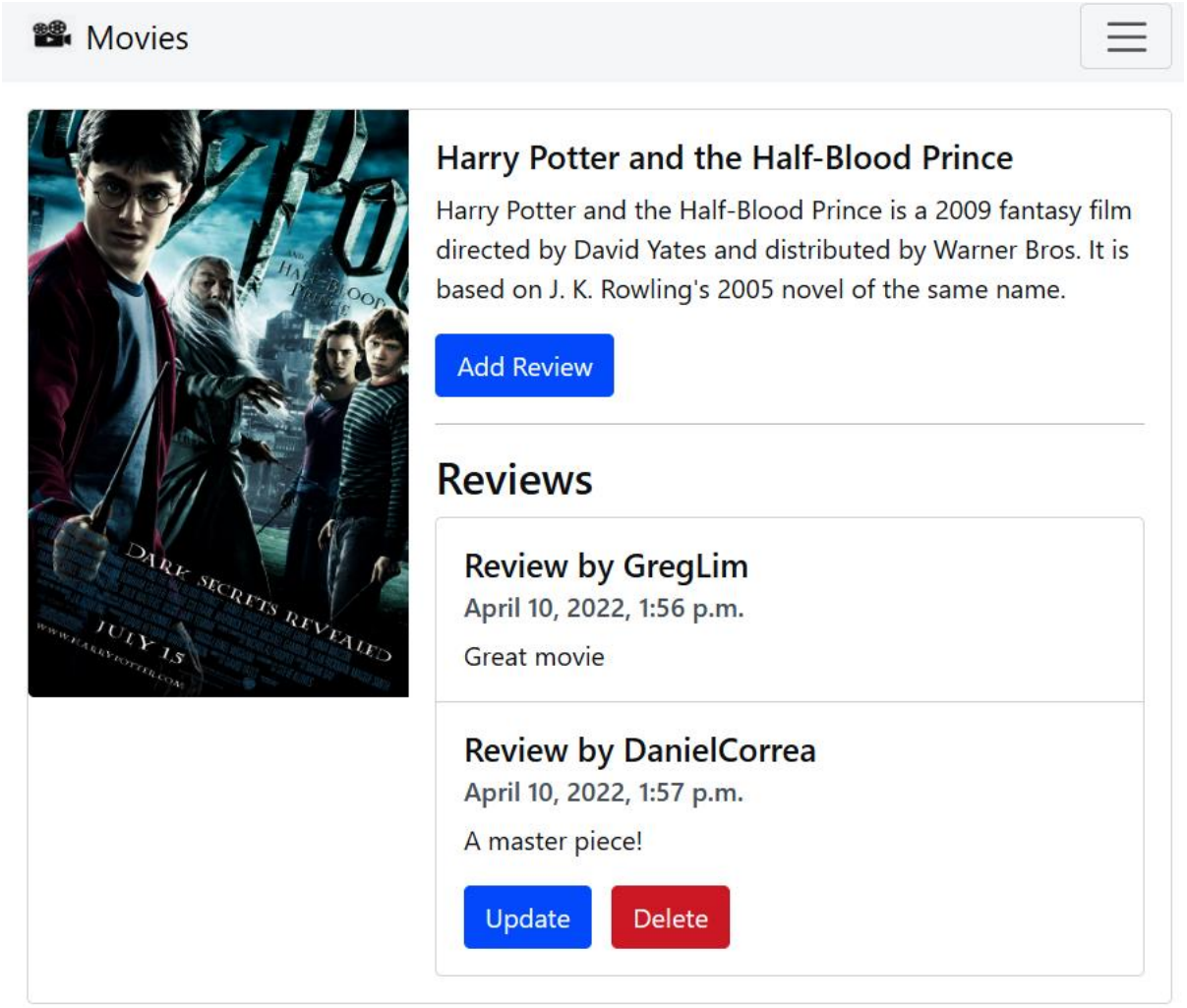
You will see the review form that Django automatically generated for you (*Figure 11.2*):



After adding the review, you can check the admin panel where it will be reflected. Next, we will see how to list reviews on the movie details page.

Listing reviews

Now, we want to list a movie's reviews on the movie details page (*Figure 11.3*):



Let's look at the steps to do so:

1. In `/movie/views.py`, in `def detail`, add the following code 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})
```

Let's see what's happening in the code. Using the `filter` function, we retrieve reviews for a particular movie only:

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

We then pass in the reviews to `detail.html`:

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

/movie/templates/detail.html

2. We list the reviews under the movie's card component (in `/movie/templates/detail.html`) 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"
              href="#">Update</a>
            <a class="btn btn-danger"
               href="#">Delete</a>
             {% endif %}
          </li>
        {% endfor %}
        </ul>
      </div>
...
```

Using a `for` loop, we render a Bootstrap list group item component for each review (`https://getbootstrap.com/docs/5.1/components/list-group/`):

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

We render `username`, the review `date` and the review `text`:

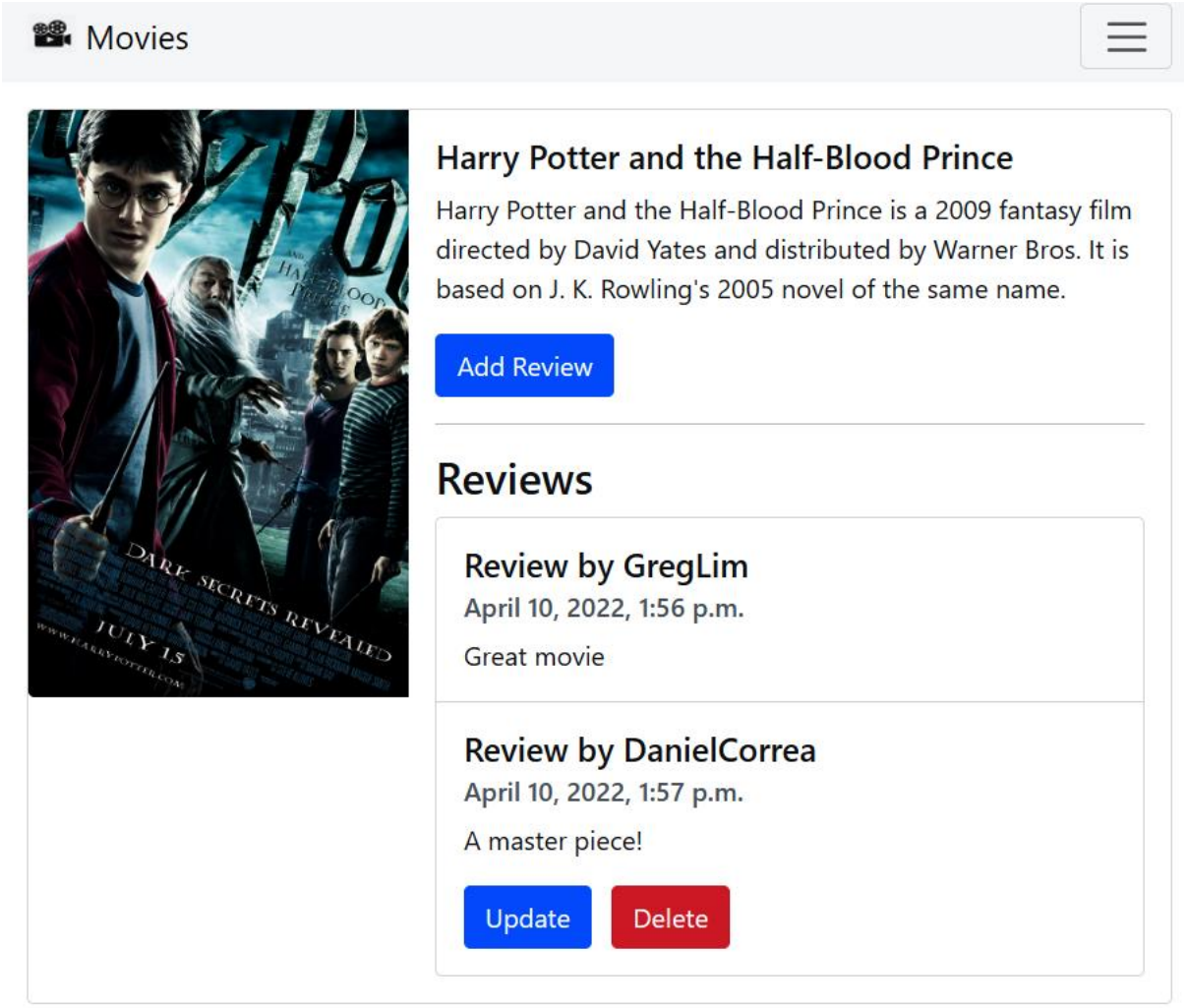
```
<h5 class="card-title">
Review by {{ review.user.username }}
</h5>
<h6 class="card-subtitle mb-2 text-muted">
{{ review.date }}
</h6>
<p class="card-text">{{ review.text }}</p>
```

We also check whether a user is logged in, and if a review belongs to the user, render the update and delete link to allow them to update/delete it:

```
{% 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 %}
```

Otherwise, we hide the update/delete links, so that a user can only update/delete reviews they have posted. They can't do so for others' reviews.

3. On 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 (*Figure 11.4*):



When you log out, you can't see them anymore. Let's continue with updating a review.

Updating a review

Let's look at the steps to do this:

1. 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 (the review's primary key) – for example, `http://localhost:8000/movie/review/2`.

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':
        return render(request, 'updatereview.html', {'review': review})
    else:
        try:
            review.text = request.POST.get('myreview')
            review.save()
            return redirect('detail', review.movie.id)
        except ValueError:
            return render(request, 'updatereview.html', {'review': review,
'error':'Bad data in form'})
```

/movie/templates/updatereview.html

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

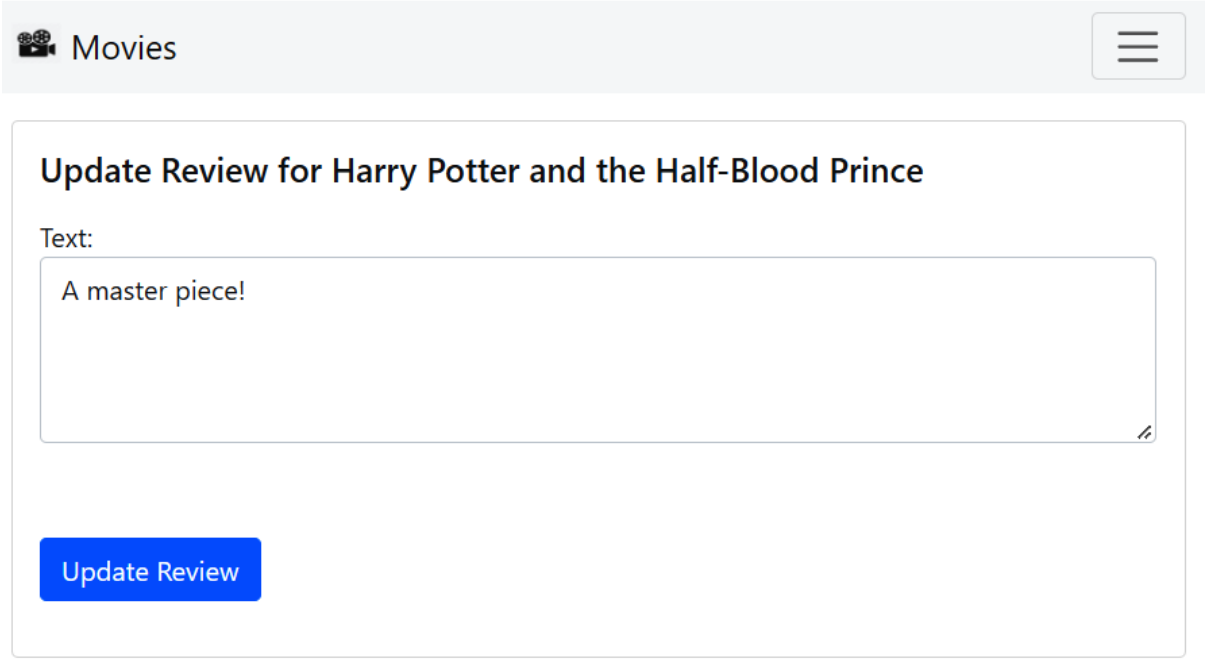
```
{% extends 'base.html' %}
{% block content %}
<div class="card mb-3">
  <div class="row g-0">
    <div>
      <div class="card-body">
        <h5 class="card-title">
          Update Review for {{ review.movie.title }}
        </h5>
        {% if error %}
        <div class="alert alert-danger mt-3" role="alert">
          {{ error }}
        </div>
        {% endif %}
        <p class="card-text">
          <form method="POST">
            {% csrf_token %}
            <div class="form-control">
                <textarea name="myreview" rows="4" cols="50">
                    {{ review.text }}
                </textarea>
            </div>
            <button type="submit" class="btn btn-primary">
              Update Review
            </button>
          </form>
        </p>
      </div>
    </div>
  </div>
</div>
{% endblock content %}
```

/movie/templates/detail.html

4. Lastly, back in `/movie/templates/detail.html`, we add the `'updatereview'` URL to the update button with the following in bold:

```
…
{% 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 %}
…
```

5. 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 (*Figure 11.5*):



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

Deleting a review

Having implemented the update of a review, let's now implement deleting a review:

1. 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'),
]
```

2. 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. As with `update`, we supply the logged-in user to ensure that only the user who created this review can delete it. We then redirect back to the movie's detail page.

3. Back in the listing of reviews on 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 %}
…
```

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

Implementing authorization

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

Currently, if a user manually enters the URL to create a review – for example, `http://localhost:8000/movie/2/create` – they can still access the form. We should authorize access to creating/updating/deleting reviews only to logged-in users. We will also authorize access to logout.

Let's look at the steps to do so:

We import and add the `@login_required` decorator to the views that we want to authorize, as shown in bold:

/movie/views.py

```
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` the following:

```
…
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 – for example, `http://localhost:8000/movie/2/create` – where you will be redirected to the login page.

12

Owned Rows, One-to-Many Model, Many-to-Many Model, Cookies, Sessions and Timezones

Owned Rows

"Owned rows" typically refer to database records or rows that are associated with a particular user or owner. This concept is often implemented in applications that have user authentication and where data needs to be segregated based on users.

For instance, in a blog application built using Django, each blog post might be associated with a specific user. These posts would be considered "owned rows" by the users who created them. This is often achieved by adding a foreign key field in the model to reference the user who owns the row.

Django One-to-Many Model:

In Django, a one-to-many relationship is a type of database relationship where each instance of one model is associated with multiple instances of another model. This is a common scenario where one entity has a collection of related entities.

For instance, consider a blog application where each blog post is written by a single author, but an author can have multiple blog posts. This is a one-to-many relationship.

Here's how you can define and use a one-to-many relationship in Django using models:

```python
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class BlogPost(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

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

Many-to-many relationships

To define a many-to-many relationship, use **ManyToManyField**.
In this example, an **Article** can be published in multiple **Publication** objects, and a **Publication** has multiple **Article** objects:

```python
from django.db import models


class Publication(models.Model):
    title = models.CharField(max_length=30)

    def __str__(self):
        return self.title


class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)

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

Django Cookies and Sessions:

In Django, cookies and sessions are used to store and manage user data across different requests and sessions. Cookies are small pieces of data stored on the user's browser, while sessions are stored on the server. Both are used to maintain user state and remember information between different interactions with a web application.

Here's how to use cookies and sessions in Django with examples:

Cookies:

Cookies are typically used to store small pieces of information on the user's browser. In Django, you can set and retrieve cookies using the `HttpResponse` object.

```python
from django.http import HttpResponse

def set_cookie(request):
    response = HttpResponse("Cookie set!")
    response.set_cookie('username', 'john_doe', max_age=3600)
    return response

def get_cookie(request):
    username = request.COOKIES.get('username', 'Guest')
    return HttpResponse(f"Hello, {username}!")
```

In the above example, the `set_cookie` function sets a cookie named 'username' with the value 'john_doe'. The `max_age` argument specifies the cookie's expiration time in seconds.

The `get_cookie` function retrieves the 'username' cookie value from the request's COOKIES dictionary and displays a personalized greeting.

Sessions

Sessions allow you to store and manage user-specific data on the server side. Django provides a built-in session framework.

```python
from django.shortcuts import render

def set_session(request):
    request.session['user_id'] = 123  # Set a session variable 'user_i
    return render(request, 'set_session.html')

def get_session(request):
    user_id = request.session.get('user_id', None)  # Retrieve the 'us
    return render(request, 'get_session.html', {'user_id': user_id})
```

In this example, the `set_session` function sets a session variable named 'user_id' to 123. The `get_session` function retrieves the 'user_id' session variable and displays it in a template.

Here's an example template (set_session.html) for the `set_session` view:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Set Session</title>
</head>
<body>
    <h1>Session Set</h1>
</body>
</html>
```

And an example template (get_session.html) for the `get_session` view:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Get Session</title>
</head>
<body>
    <h1>User ID: {{ user_id }}</h1>
</body>
</html>
```

In the above examples, when a user visits the `set_session` view, a session variable 'user_id' is set. When they visit the `get_session` view, the session variable is retrieved and displayed.

Remember that session data is stored on the server side, typically in a database or cache.

In your project's `settings.py`, you can set the session timeout to 60 seconds by adding the following line:

```
SESSION_COOKIE_AGE = 60
```

Timezone

To change the time zone to Indian Standard Time (IST) in a Django project, follow these steps:

1. **Settings.py Configuration:** Open your Django project's `settings.py` file.
2. **Import Timezone:** At the top of the `settings.py` file, add the following import:

   ```
   from django.utils import timezone
   ```
3. **Time Zone Setting:** Locate the `TIME_ZONE` setting in your `settings.py` file. It's usually set to `'UTC'` by default. Change it to `'Asia/Kolkata'`, which corresponds to the Indian Standard Time zone.

   ```
   TIME_ZONE = 'Asia/Kolkata'
   ```