



Building Websites

with

Django

Build and Deploy Professional Websites with Python Programming
and the Django Framework



AWANISH RANJAN

bpb

Building Websites with Django

*Build and Deploy Professional Websites with
Python Programming and the Django Framework*

Awanish Ranjan



www.bpbonline.com

FIRST EDITION 2021

Copyright © BPB Publications, India

ISBN: 978-93-89328-288

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

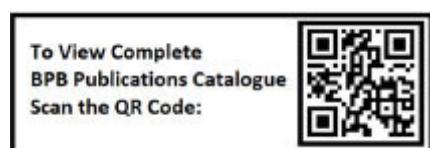
Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747



Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

*Rajan and Suman Singh
My parents, source of my courage*

About the Author

After completing a Bachelor of Engineering in 2015, **Awanish** started working with the largest IT company in the country. In the tenure of five years, Awanish worked with and learned from highly experienced software developers. Awanish was creating world-class solutions for clients in the US, UK, Canada, Central Europe, and APAC.

Awanish started his career in Java but soon realized the potential of Python and how easily the shortcomings of other languages were resolved in Python. He began with scripting, creating inhouse ETL tools, and then web applications using Django and now is building Content Management Systems or popularly known as Django CMS.

In his free time, Awanish likes to teach and has been providing classroom as well as online training to students. He would love to share his learnings and experience.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my wife for putting up with me while I was spending many weekends and evenings on writing—I could have never completed this book without her support.

I want to thank people at Django Software Foundation for creating and maintain the official Django documentation, which would help anyone working on Django. Thanks to Heroku for providing a free hosting platform.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them. This book would not have happened if I had not had the support from the publishers. They were with me when the schedule extended.

Preface

Django is a Python-based web framework. It gained popularity because it provides features like speedy development, structured programming flow, and it does most of the heavy lifting hence needs less learning. This attracts programmers and users who want to build websites quickly.

In this book, you read about the journey of developing web applications using Django. You will start with the basics of creating a Django project. Then you will read about different components of Django in detail so that you have enough theoretical knowledge to create websites of your own. Later you will be creating a Django web application from scratch and deploy it on a public url, which you can share with your friends.

After reading this book, you will have a good understanding of how Django projects work and how to create practical web applications. The primary goal of this book is to provide information and skills that are necessary to build backend of a Django application deploy it on a publicly accessible url. There are theoretical chapters that give you sufficient information about different features of the components of a Django project. This enables you to think beyond the sample project created at the end of the book.

Over the 16 chapters in this book, you will learn the following:

[Chapter 1](#) introduces the concept and need for frameworks. You will read about the advantages of using the Django Framework and the concept of virtual environments. You will also create a dummy project.

[Chapter 2](#) discusses the architecture of Django. It explains the significance of different components of the architecture.

[Chapter](#) with a dummy project up and running, you will read about the settings needed to be configured in a Django project. The importance of the `settings.py` file and the configuration options provided is discussed in this chapter.

[Chapter 4](#) is a key chapter that discusses, in-depth, the built-in admin module offered by Django. You will read about the steps to set up an admin module and how to use the admin module to manipulate data in the project database.

[Chapter 5](#) discusses one of the key concepts needed in any framework – how to interact with the database programmatically. You will read about using query sets to manipulate data in the project database programmatically.

[Chapter 6](#) is all about working on the different components of the Django mini-project. You will be working on models, views, URLs, and templates. In this chapter, you will have an overview of what and how the different components work.

[Chapter 7](#) begins the detailed theory section of the book. This chapter focuses on Models. You will read about models, model fields, model fields options, etc. You will read about the relationship between the models.

[Chapter 8](#) focuses on views. You will read about different types of views and when to use them. You will read about the function and class-based views.

[Chapter 9](#) describes how to create templates. You will read about the Django Templating Language in brief. The template tags and their usage is discussed in this chapter.

[Chapter 10](#) describes how to write url-patterns. You will read about different functions available to write url-patterns and how to use rex-ex to write the string patterns.

[Chapter 11](#) describes how to take input from users using forms. You will read about creating form and manipulating the data received from forms. You will also read about how to create forms by using models.

[Chapter 12](#) begins the project which you will be deploying. In this chapter, you will create a new project and set it up. You will read about the basic configurations needed in a major project.

[Chapter 13](#) is about creating an accounts app which will provide user management features in the project. You will read about how to use the built-in user model to enable user registration, login,

and logout functionalities. You will read about how to set up authorization in your project.

[Chapter](#) again, is about creating an app in Django. This app will provide features like segregation and will help in implementing authorizations.

[Chapter 15](#) discusses creating an app that will be used to manage content on your website.

[Chapter 16](#) discusses the process of deploying your application on a server with publicly accessible url.

***Downloading the
coloured images:***

Please follow the link to download the
Coloured Images of the book:

<https://rebrand.ly/68914>

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB IS SEARCHING FOR AUTHORS LIKE YOU

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. What is Django?

Introduction

Structure

Objectives

Django overview

What is a framework?

Why do we need a framework?

What are some famous web frameworks?

Is learning Django worth it?

What is a virtual environment?

How to create and use a virtual environment?

How to install Django?

How to create a Django project?

Overview of your Django project files

Conclusion

Questions

2. Overview of MVT Architecture

Introduction

Structure

Objectives

What is architecture?

What is MVT?

What are models in Django's MVT?

What are Views in Django's MVT?

What are Templates in Django's MVT?

Conclusion

Questions

3. Understanding Django Settings

Introduction

Structure

Objectives

Exploring some common settings

Core settings

Authentication

Static files

Conclusion

Questions

4. The Django Admin Utility

Introduction

Structure

Objectives

Admin page of your Django project

Creating an app

Editing models.py, settings.py, and admin.py.

Adding data to your database

Editing data in your database

Conclusion

Questions

5. Interacting with the Database Using Query Sets

Introduction

Structure

Objectives

ORM overview

Query sets

Adding elements to your database

Manipulating elements of your database

Deleting elements of your database

Conclusion

Questions

6. Enhancing Your Project

Introduction

Structure

Objectives

Understanding requirements

Making Models

Creating views

Configuring urls.py

Making templates

Testing

Conclusion

Questions

7. Understanding Models

Introduction

Structure

Objectives

Introduction to models

Model fields

Field options

Meta options

Model methods

Relationship between models

[Connecting models](#)

[Conclusion](#)

[Questions](#)

[8. Django Views](#)

[Structure](#)

[Objectives](#)

[Introduction](#)

[Types of views](#)

[Function-based views](#)

[Class-based views](#)

[When to use class and function-based views](#)

[Built-in class-based views](#)

[Base view](#)

[View](#)

[Templateview](#)

[RedirectView](#)

[Generic views](#)

[List view](#)

[Detail view](#)

[Conclusion](#)

[Questions](#)

[9. Django Templates](#)

[Structure](#)

[Objectives](#)

[Introduction](#)

[Configuration](#)

[Template inheritance](#)

[Django Templating Language](#)

[Syntax](#)

[Conclusion](#)

[Questions](#)

[10. URLs and Regex](#)

[Structure](#)

[Objectives](#)

[Introduction](#)

[Functions available in URLconfs](#)

[Regex](#)

[Writing a regex for different url-functions](#)

[url/re_path\(\)](#)

[path\(\)](#)

[Conclusion](#)

[Questions](#)

[11. Forms in Django](#)

[Structure](#)

[Objectives](#)

[Introduction](#)

[Building basic forms](#)

[Fetching data entered in the forms](#)

[Form fields and arguments](#)

[Form field arguments](#)

[Form fields](#)

[Form validation](#)

[Model forms](#)

[Conclusion](#)

[Questions](#)

12. Setting Up Project

Structure

Objectives

Introduction

Setting up the project

Updating the settings.py options

Testing

Enhancing the project

Creating superuser

Conclusion

Questions

13. The Account App

Introduction

Structure

Objective

Difference between authentication and authorization

Django's built-in authentication module - auth

The User model of the auth module

The PermissionMixin model of the auth module

models.py

forms.py

views.py

urls.py

settings.py

Templates

Testing

Conclusion

Questions

14. The Genre App

Introduction

Structure

Objective

models.py

admin.py

views.py

urls.py

Templates

Testing

Conclusion

Questions

15. The Posts App

Introduction

Structure

Objective

models.py

views.py

forms.py

urls.py

Templates

Testing

Conclusion

Questions

16. Deploying the Website

Introduction

Structure

Objective

[Deployment configurations in the project](#)

[Setting up Heroku](#)

[Conclusion](#)

[Questions](#)

[Index](#)

CHAPTER 1

What is Django?

Introduction

Python is one of the most versatile programming languages used in the industry today. Because of Python's simplicity, a programmer gets to focus more on the solution to the problem than on the solution to be implemented. Since it is open-source, we have a huge collection of free and open-source libraries, frameworks, modules, etc. available to use. Django is one of them. Before moving forward, you must have an understanding of frameworks and what Django can offer so that the outcomes match your expectations.

In short, Django is a framework build on Python and it is used for web application development. It is available for free and is open source. You will learn about the basic concepts of a framework. You will know what a framework is and see how Django provides a solution to your web application development requirements.

Structure

What is a framework?

Why do we need a web framework?

What are some famous web frameworks?

Is learning Django worth it?

What is a virtual environment?

How to create and use a virtual environment?

How to install Django?

How to create a Django project?

Overview of your Django project files

Objectives

Understanding the concept of frameworks.

Understanding the need for frameworks.

Familiarizing with the benefits of Django.

Understanding the concept of virtual environments.

Learning to install Django.

Creating a Django project

Understanding of the file system of Django

Django overview

Django is famous for its ability to create web applications rapidly. How does this happen? Django has inbuilt middleware and other stuff needed to run a web application. All you need to do is focus on your application. Django very elegantly provides the settings and connections in a configurable manner. You can simply create your applications and configure their settings as per your requirement and you are good to go.

Django was developed in 2003 by web developers at *Lawrence Journal-World* newspaper, *Adrian* and *Simon*. It was released publicly under a BSD license in July 2005. The framework is named after the guitarist *Django*. Now, the Django Software Foundation has been maintaining Django. This official website for Django is [This](#) is where you can get all the latest updates about Django.

The best thing about Django is that it is very elegantly documented. It is easy to learn and implement. You can find the latest documentation at [Currently, as of September 2019, Django 2.2 is the latest version. The other widely used version is Django 1.8.](#) I would recommend that you stick to the latest version.

What is a framework?

Let us suppose you have a simple task of drawing a square of side measuring 5cm. You can take your instruments, measure 5cms, and draw your square. Now, if you are asked to draw 1000 such squares, will you follow the same process, not likely? You would create a square frame of the given measurements and use that frame to draw the 1000 squares.

A software framework is very similar to this. Software development has seen a significant history and the requirements of a kind of software are pretty much the same. In a nutshell, a web application needs authentication and authorization, homepage, database connectivity, HTML pages for frontend, CSS for styling, a URL mapper, and a controller to choose what action is to be performed based on the request, etc.

These needs of a software developer or in this case, a web application developer has been identified and the setup has been created (like the frame in the squares example) where a web app developer can easily configure the settings (like the measurements in the squares example) and get the job done quickly and with ease.

Why do we need a framework?

The understanding of the framework makes it pretty much clear why we use frameworks. Let's quantify the advantages of using a framework:

It sets an industry-recognized software design structure. If every developer uses different designs, then it becomes difficult for the software owner or the client to maintain the software once the initial developer leaves the project. Thus, using a framework ensures that a standard design is followed.

Re-writing code for common functionalities that are already available in frameworks does not make sense. Also, frameworks allow us to modify the functionalities as per our needs. So, customization is easy.

The frameworks have been present for quite some time and extensively tested by developers worldwide, issues are reported, and bug fixes are continuously rolled. This ensures the software is bug-free.

The frameworks keep updating with new features as the technology advances. Using a framework keeps your software updated. If you choose not to use a framework, then it becomes the developer's responsibility to keep the software updated with the latest technology advancements.

Now, you are acquainted with what frameworks are and why we use it. Let's take a look at what web frameworks are available in the market exception.

What are some famous web frameworks?

Apart from Django, there are several other web application frameworks available in the industry which you should be aware of which are as follows:

Flask: Flask is a web application framework written in Python. Flask is based on the WSGI toolkit and Jinja2 template engine.

Ruby on Rails: Ruby on Rails is a productive web application framework based on Ruby. One can develop an application at least quite faster with Rails than a typical Java framework.

Angular: Angular is a framework on JavaScript by Google which helps us in building powerful web apps. It is a framework to build large scale and high-performance web applications while keeping them as easy-to-maintain.

ASP.NET: ASP.NET is a framework developed by Microsoft, which helps us to build robust web applications for PCs, as well as mobile devices. It is a high-performance and lightweight framework for building web applications using .NET.

Spring: Spring is the most popular application development framework for enterprise Java. Developers around the globe use Spring to create high-performance and robust web apps.

PLAY: Play is one of the modern web application frameworks written in Java and Scala. It follows the MVC architecture and aims to optimize the developer's productivity by using convention over configuration, hot code reloading, and display of errors in the browser.

Other frameworks worth mentioning are Laravel, Symfony, CodeIgniter, Express, React.js, Node.js, Hibernate, etc.

Is learning Django worth it?

Before you dive into Django and dedicate your time and effort to it, you should ask if it's worth it. Is it used in industry? You will be glad to know that some of the most famous websites use Django. Examples: Disqus, Instagram, Knight Foundation, MacArthur Foundation, Mozilla, National Geographic, Open Knowledge Foundation, Pinterest, Open Stack, etc. So yes, learning Django is completely worth it. The best part is the more learn, the more it grows over you.

What is a virtual environment?

While developing software or web applications, you will be using a lot of pre-written code legally available from different sources. These sets of code are called libraries, packages, add-ons, plugins, etc. If you are using a framework, then that framework will also have a huge set of pre-written code which will help you implement different features in your application. To use these pre-written codes, you will need to install the libraries, plugins, etc. in your project/application.

The catch here is that different applications are expected to have different features. So, to implement those different features, you will need different set of frameworks, libraries, add-ons, plugins, etc.

Let's assume you are developing two different applications on your personal computer. *Project 1* is using Django 2.1, Python 3.5, bootstrap 4, etc. Whereas *Project 2* is using Flask, Python 2.7, bootstrap 3, etc. Now, you have a requirement to install both (3.5 and 2.7) the versions on Python, Django, Flask, different versions of bootstrap, etc. When you run your projects, the interpreter will be confused about which version of the Python to use. For *Project 1*, installation of *Project 2* will be of no use and will create conflicts and vice-versa for *Project 2*.

To avoid such conflicts and confusions, we use virtual environments. We create a setup that takes some space on the server or your computer and use it for all the installations related to a project. When you work on that project, you just need to activate the corresponding setup and the interpreter uses the installations in that setup only. This setup is called a **virtual**

Different setups or virtual environments are created for different projects. Therefore, there is no conflict between the installations of one project with another. You can run different projects simultaneously using virtual environments. All you need to do is to set the path of an interpreter to the correct virtual environment. All your projects will use their different interpreters and will run without any conflicts.

How to create and use a virtual environment?

So now you have an overview of virtual environments. Let's create one and see how it looks like. You will need an active internet connection.

Install Python:

Let's begin by installing Python. Go to python official website - and then go to the **Downloads** tab.

Select your operating system (In this book, we will be using Windows.):

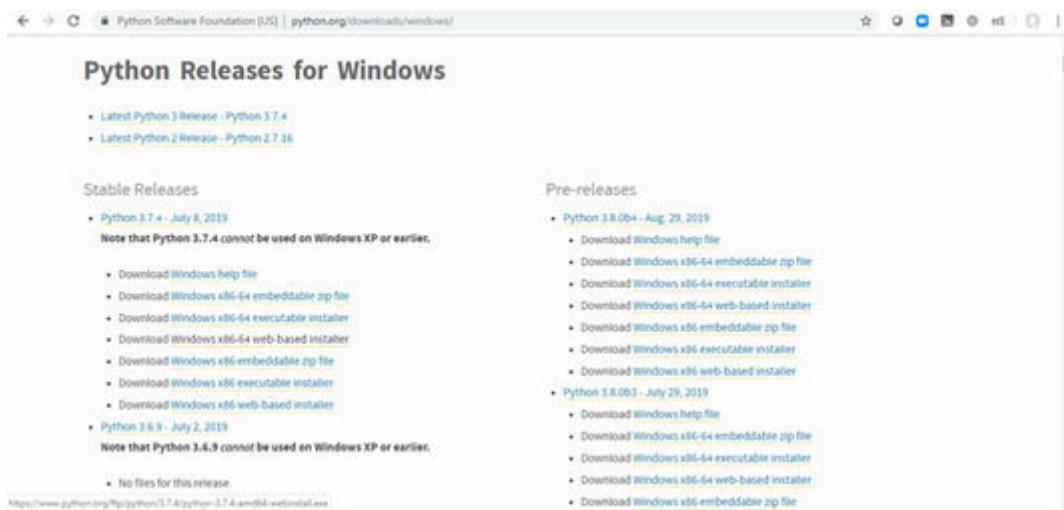


Figure 1.1: Operating system choices for python installation

Select your operating system and download the Python installable:

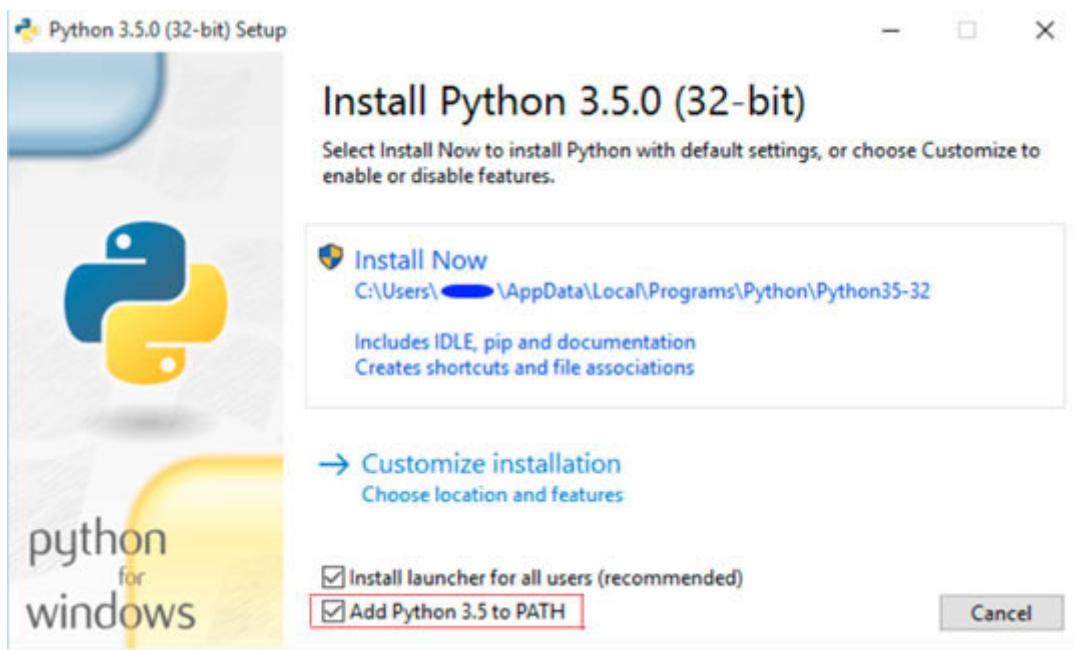


Figure 1.2: Select Add Python to the PATH checkbox

Once downloaded, run the installable file. You will see a similar window. Choose your location for the installation and make sure that you checkmark the last checkbox **Python x.x to This**. This will configure your operating system to use this Python installation for executing Python programs as default if any other installations are not mentioned.

Installing a virtual environment:

Open your Windows Power Shell or CMD.

Run the install command without the quotes.

This will install the virtual environment on your machine.

If you face errors like pip not you need to install Or you can uninstall Python and again install it by following the preceding steps.

Creating a virtual environment:

Create a root folder for your project. A folder that will contain everything related to the project. Let's call it

Change the directory to Project_root in PowerShell or CMD or your Linux terminal and run the command without the quotes. Here, virtualenv is the command used to create a virtual environment and prenv is the name of your virtual environment. You can choose any name for your virtual environment, although it is recommended to keep it similar to the name of the project so that one can easily relate the virtual environment to its corresponding project:

```
04 Command Prompt
Microsoft Windows [Version 10.0.17763.678]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\awani>cd C:\Work\Project_root

C:\Work\Project_root>virtualenv prenv
Using base prefix 'c:\\users\\awani\\appdata\\local\\programs\\python\\python37-32'
New python executable in C:\\Work\\Project_root\\prenv\\Scripts\\python.exe
Installing setuptools, pip, wheel...
done.
```

Figure 1.3: Creating a virtual environment

Your virtual environment is now ready. Let's explore it and see what it has to offer. Run the command without the quotes:

```
C:\Work\Project_root>cd prenv  
C:\Work\Project_root\prenv>dir  
Volume in drive C is Windows  
Volume Serial Number is 4883-E662  
  
Directory of C:\Work\Project_root\prenv  
  
03-09-2019  21:02    <DIR>          .  
03-09-2019  21:02    <DIR>          ..  
07-06-2019  18:17    <DIR>        Include  
03-09-2019  21:02    <DIR>        Lib  
25-03-2019  21:31                30,195 LICENSE.txt  
03-09-2019  21:02    <DIR>        Scripts  
03-09-2019  21:02    <DIR>        tcl  
                           1 File(s)      30,195 bytes  
                           6 Dir(s)   175,674,355,712 bytes free
```

Figure 1.4: Virtual environment folder

Now, run the dir command to see the contents of your virtual environment folder. You can see the folders like Include, Lib, Scripts, and the LICENSE.txt file. The Lib folder contains all your library files and installations. The Scripts folder has the scripts which perform certain specific tasks. The Include folder contains the interpreter files.

Now, once you have your virtual environment ready, you need to activate it. It is activated and deactivated by scripts present in the Scripts folder. Run the command without the quotes. Then, run without the quotes:

```
C:\Work\Project_root\prenv>cd Scripts  
C:\Work\Project_root\prenv\Scripts>activate  
(prenv) C:\Work\Project_root\prenv\Scripts>
```

Figure 1.5: Activating a virtual environment

Your virtual environment is active now. The indication of activation or deactivation of any virtual environment is the name of the virtual environment at the beginning of the line in the terminal.

Let's install the Python package and check whether this works. Run the pip install numpy command. Once the installation completes, go to your Lib/site-packages folder and check for numpy installations.

This is how you can install packages and libraries in your virtual environments:

```
C:\Work\Project_root\prenv\Scripts>activate  
(prenv) C:\Work\Project_root\prenv\Scripts>pip install numpy  
Collecting numpy  
  Downloading https://files.pythonhosted.org/packages/c3/13/a991b874825a195  
aefb9cf53a1a632099622237d8701dbd4a18804fa5144/numpy-1.17.1-cp37-cp37m-win32  
.whl (10.8MB)  
|██████████| 10.8MB 46kB/s  
Installing collected packages: numpy  
Successfully installed numpy-1.17.1  
(prenv) C:\Work\Project_root\prenv\Scripts>
```

Figure 1.6: Checking installations

Now, any installations made using this terminal will be confined to this environment only. Any projects that run from the current terminal will use the installations in this environment.

If you have a project anywhere on your server or computer and you want to execute it using the installations of the environment, then simply activate this environment as shown earlier and then change the directory to your project directory and run your project:

```
C:\Work\Project_root\prenv>cd Scripts  
C:\Work\Project_root\prenv\Scripts>activate  
(prenv) C:\Work\Project_root\prenv\Scripts>deactivate  
C:\Work\Project_root\prenv\Scripts>
```

Figure 1.7: Deactivating the virtual environment folder

To deactivate this environment, run the deactivate command.

Now, you have a virtual environment installed in your project and you are good to go. Let's install Django and move one step closer to create your web application.

How to install Django?

In the previous section, you learned about virtual environments and how to use them. Now you have a fair understanding of how to use the terminal to install packages in virtual environments. Now, in this chapter, you will use the virtual environment created earlier and install Django on it.

We used pip twice in the previous sections. You must be wondering what is a pip? Well, pip is like an official Python package installer. If there is any validated Python library or package present at then pip can be used to install it. PIP gets installed by default with Python 3 installation. If you don't have pip in your system, then you can install pip by following the instructions at

Let us get back to installing Django. Go back to your CMD window or terminal and activate your virtual environment. While installing any package, you don't need to be present at any particular location. Just the right virtual environment should be active.

Now, run the pip install django command. This will install the latest version of Django on your virtual environment:

```
(prenv) C:\Work\Project_root>pip install django
Collecting django
  Downloading https://files.pythonhosted.org/packages/94/9f/a56f7893b1
280e5019482260e246ab944d54a9a633a01ed04683d9ce5078/Django-2.2.5-py3-no
ne-any.whl (7.5MB)
    |████████| 7.5MB 45kB/s
Collecting pytz (from django)
  Downloading https://files.pythonhosted.org/packages/87/76/46d697698a
143e05f77bec5a526bf4e56a0be61d63425b68f4ba553b51f2/pytz-2019.2-py3
-none-any.whl (508kB)
    |████████| 512kB 52kB/s
Collecting sqlparse (from django)
  Using cached https://files.pythonhosted.org/packages/ef/53/900f7d2a5
4557c6a37886585a91336520e5539e3ae2423ff1102daf4f3a7/sqlparse-0.3.0-py2
.py3-none-any.whl
Installing collected packages: pytz, sqlparse, django
Successfully installed django-2.2.5 pytz-2019.2 sqlparse-0.3.0
(prenv) C:\Work\Project_root>
```

Figure 1.8: Installing Django

If you wish to install any other version of Django, then use the pip install Django==x.x.x command where x.x.x is the version number. Let's verify the Django installation. Go to Lib/site-packages of your virtual environment and look for Django installations:

```
This PC > Windows (C:) > Work > Project_root
(prenv) C:\Work\Project_root>cd C:\Work\Project_root\prenv\Lib\site-packages
(prenv) C:\Work\Project_root\prenv\Lib\site-packages>dir
Volume in drive C is Windows
Volume Serial Number is 4883-E662

Directory of C:\Work\Project_root\prenv\Lib\site-packages

03-09-2019 22:34    <DIR>      .
03-09-2019 22:34    <DIR>      ..
03-09-2019 22:34    <DIR>      django
03-09-2019 22:34    <DIR>      Django-2.2.5.dist-info
03-09-2019 21:02    <DIR>      126 easy_install.py
03-09-2019 21:42    <DIR>      numpy
03-09-2019 21:02    <DIR>      numpy-1.17.1.dist-info
03-09-2019 21:02    <DIR>      pip
03-09-2019 21:02    <DIR>      pip-19.2.3.dist-info
03-09-2019 21:42    <DIR>      pkg_resources
03-09-2019 21:02    <DIR>      pytz
03-09-2019 21:02    <DIR>      pytz-2019.2.dist-info
03-09-2019 21:02    <DIR>      setuptools
03-09-2019 21:02    <DIR>      setuptools-41.2.0.dist-info
03-09-2019 21:02    <DIR>      sqlparse
03-09-2019 21:02    <DIR>      sqlparse-0.3.0.dist-info
03-09-2019 22:34    <DIR>      wheel
03-09-2019 21:02    <DIR>      wheel-0.33.6.dist-info
03-09-2019 21:02    <DIR>      __pycache__
03-09-2019 21:02    <DIR>      126 bytes
1 File(s)           126 bytes
18 Dir(s)   175,468,957,696 bytes free

(prenv) C:\Work\Project_root\prenv\Lib\site-packages>
```

Figure 1.9: Exploring Django directories

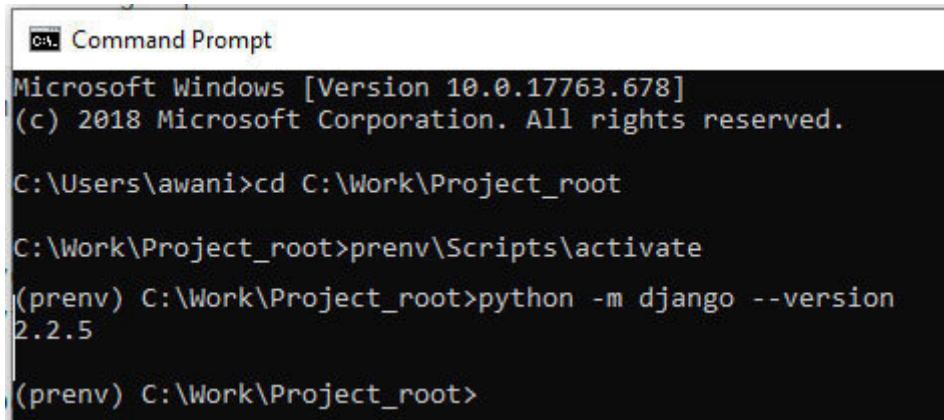
Django is installed on your system. That was pretty easy, right!

How to create a Django project?

In the previous section, we installed Django in our virtual environment. Now, we are going to start building our web application. Before we move forward, it is important to understand the difference between a project and an application in context with Django. In general, an application is something that performs a certain task. Like a music player web application plays music. Although when you open any music player web application, you will find features like playlists, popular songs, etc. If you look at this music player in context with Django, these little features are separate applications that are clubbed together in a project called **Music Player**.

So, in Django, we have a project which consists of one or more, small or big applications integrated. All these applications work together and perform the expected task. Let's create a Django project now. Open CMD, activate your virtual environment, and change the directory to your project_root folder.

Now, to check whether you have Django installed in your venv or the version of Django that is installed, run the `python -m Django --version` command. This will show you the version of Django that is installed in your venv. Here, I have 2.2.5 version of Django installed:



```
Command Prompt
Microsoft Windows [Version 10.0.17763.678]
(c) 2018 Microsoft Corporation. All rights reserved.

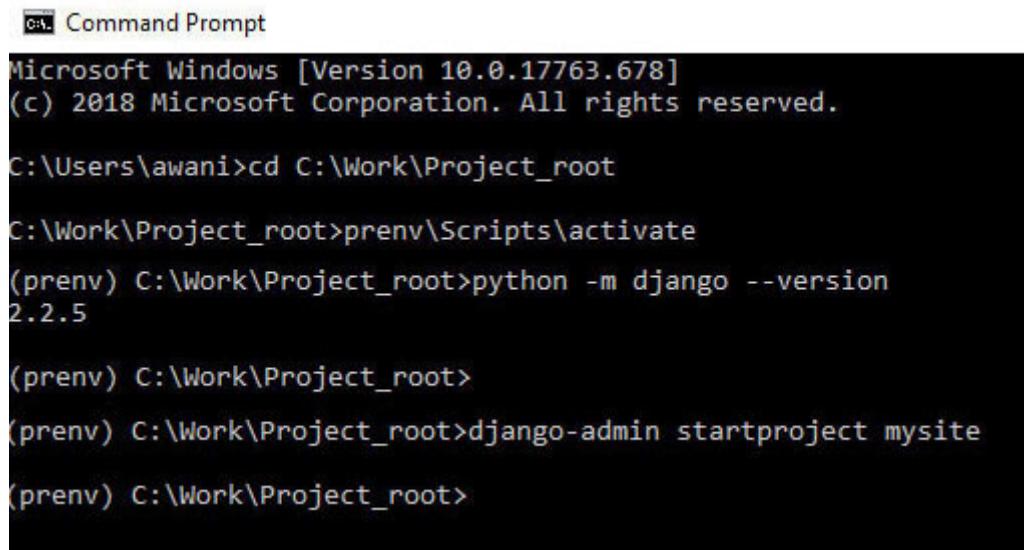
C:\Users\awani>cd C:\Work\Project_root

C:\Work\Project_root>prenv\Scripts\activate
(prenv) C:\Work\Project_root>python -m django --version
2.2.5

(prenv) C:\Work\Project_root>
```

Figure 1.10: Activating a virtual environment

Now, run the django-admin startproject mysite command. This will create a project name mysite in your Project_root directory. Here, mysite is the name of the project:



```
Command Prompt
Microsoft Windows [Version 10.0.17763.678]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\awani>cd C:\Work\Project_root

C:\Work\Project_root>prenv\Scripts\activate
(prenv) C:\Work\Project_root>python -m django --version
2.2.5

(prenv) C:\Work\Project_root>
(prenv) C:\Work\Project_root>django-admin startproject mysite
(prenv) C:\Work\Project_root>
```

Figure 1.11: Creating a Django project

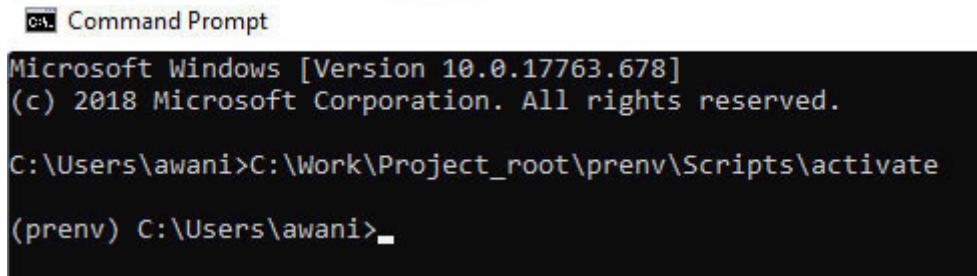
The project mysite is the most ideal for an introduction to Django project as suggested by the Django Software Foundation and is well documented on the Django official website. We will take this

project to introduce the different components of Django in the first section of the book. Later, you will build your own web application and deploy it.

Overview of your Django project files

Your project is set and ready to be tested. Follow the given steps to test the work done so far:

Open your terminal and activate your virtual environment.



```
Command Prompt
Microsoft Windows [Version 10.0.17763.678]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\awani>C:\Work\Project_root\prenv\Scripts\activate
(prenv) C:\Users\awani>
```

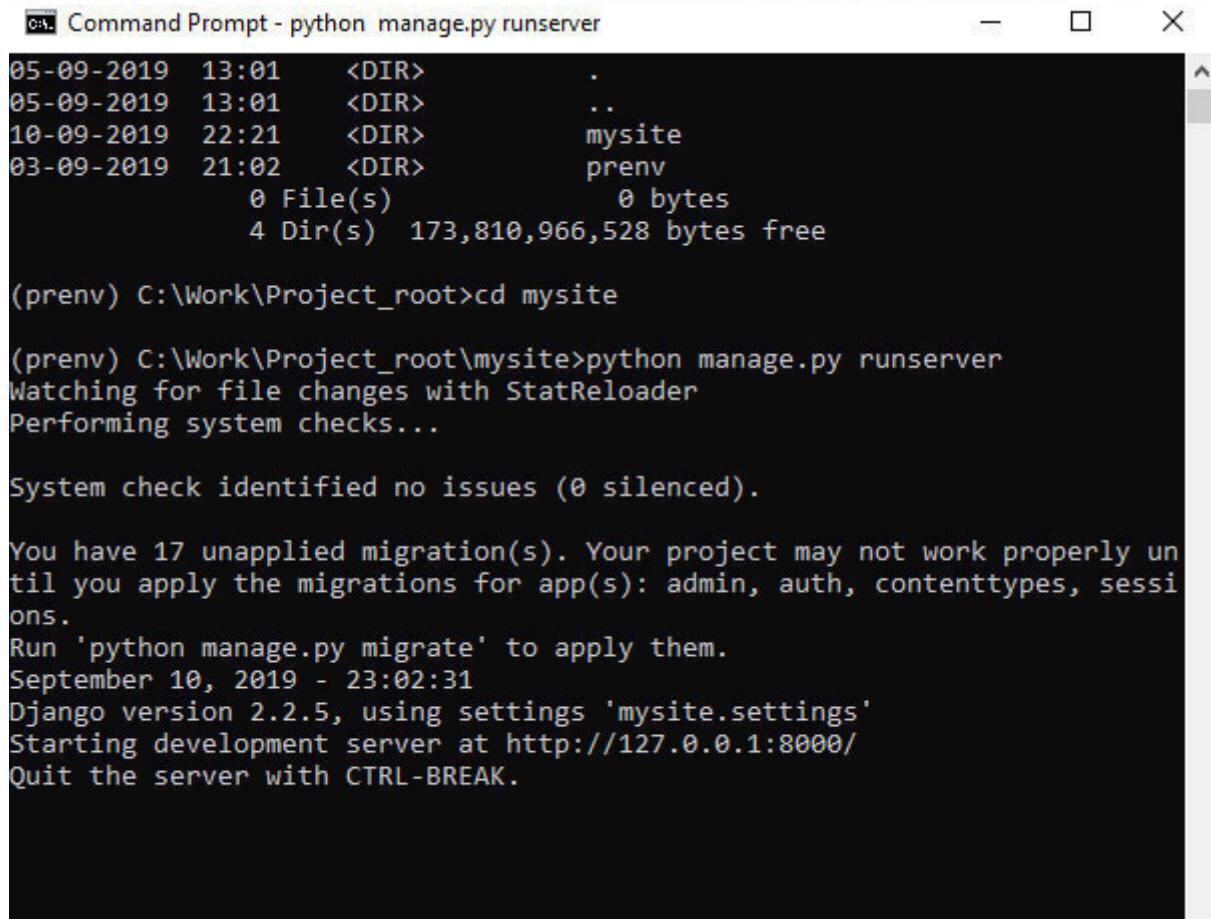
Figure 1.12: Activating virtual environment

Change the directory to your Project_root folder and the command

```
 Command Prompt  
Microsoft Windows [Version 10.0.17763.678]  
(c) 2018 Microsoft Corporation. All rights reserved.  
  
C:\Users\awani>C:\Work\Project_root\prenv\Scripts\activate  
  
(prenv) C:\Users\awani>cd C:\Work\Project_root  
  
(prenv) C:\Work\Project_root>dir  
Volume in drive C is Windows  
Volume Serial Number is 4883-E662  
  
Directory of C:\Work\Project_root  
  
05-09-2019  13:01    <DIR>          .  
05-09-2019  13:01    <DIR>          ..  
10-09-2019  22:21    <DIR>          mysite  
03-09-2019  21:02    <DIR>          prenv  
                      0 File(s)           0 bytes  
                      4 Dir(s)  173,810,966,528 bytes free  
  
(prenv) C:\Work\Project_root>
```

Figure 1.13: Exploring the Project folder

Here, you can see two folders: your virtual environment folder and your Django Project folder. The Project_root folder is the base folder. The prenv folder contains all the installations required for the Django project. The mysite folder is the actual project folder which will contain all your project-related files and code that you have written. Change the directory to your Django project folder. Now, execute the python manage.py runserver command. This command will initiate a server at the default host and port:



```
Command Prompt - python manage.py runserver
05-09-2019 13:01    <DIR>      .
05-09-2019 13:01    <DIR>      ..
10-09-2019 22:21    <DIR>      mysite
03-09-2019 21:02    <DIR>      prenv
          0 File(s)        0 bytes
          4 Dir(s)  173,810,966,528 bytes free

(prenv) C:\Work\Project_root>cd mysite

(prenv) C:\Work\Project_root\mysite>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly un
til you apply the migrations for app(s): admin, auth, contenttypes, sessi
ons.
Run 'python manage.py migrate' to apply them.
September 10, 2019 - 23:02:31
Django version 2.2.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Figure 1.14: Starting the development server

Now, open the server in the browser. By default, it will be This is your development server. If the following screen comes up, it means you have correctly executed all the steps:

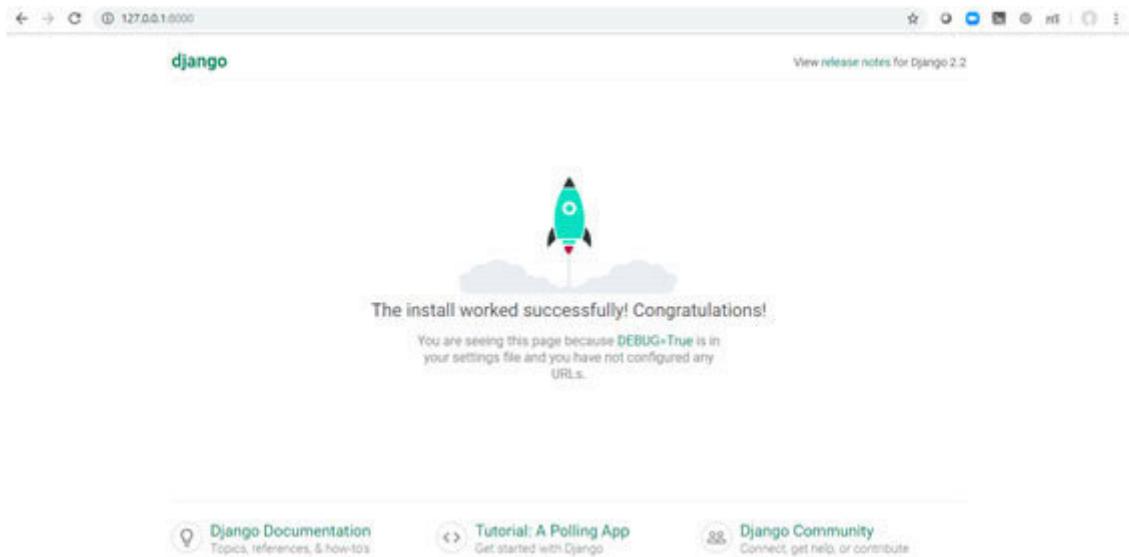
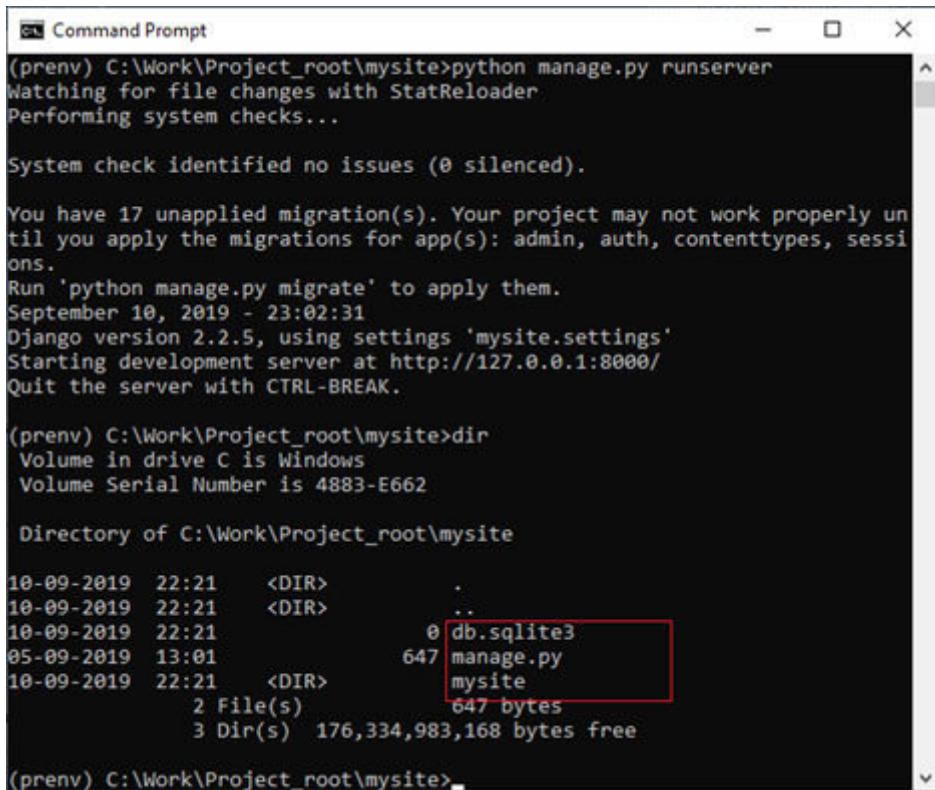


Figure 1.15: Opening the server in a browser

Congratulations! Your project is working fine. Let's dig in and see what we have.

For now, stop your server by pressing *Ctrl + C* together. You need to be in your Django project folder. Run the `dir` command. You will see the following items:



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "python manage.py runserver". The output indicates that the server is running at http://127.0.0.1:8000/. A "dir" command is then run, listing the contents of the current directory (mysite). The database file "db.sqlite3" and the management script "manage.py" are highlighted with a red rectangle.

```
(prenv) C:\Work\Project_root\mysite>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly un
til you apply the migrations for app(s): admin, auth, contenttypes, sessi
ons.
Run 'python manage.py migrate' to apply them.
September 10, 2019 - 23:02:31
Django version 2.2.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

(prenv) C:\Work\Project_root\mysite>dir
Volume in drive C is Windows
Volume Serial Number is 4883-E662

Directory of C:\Work\Project_root\mysite

10-09-2019  22:21    <DIR>      .
10-09-2019  22:21    <DIR>      ..
10-09-2019  22:21                0 db.sqlite3
05-09-2019  13:01            647 manage.py
10-09-2019  22:21    <DIR>      mysite
                           2 File(s)       647 bytes
                           3 Dir(s)  176,334,983,168 bytes free

(prenv) C:\Work\Project_root\mysite>
```

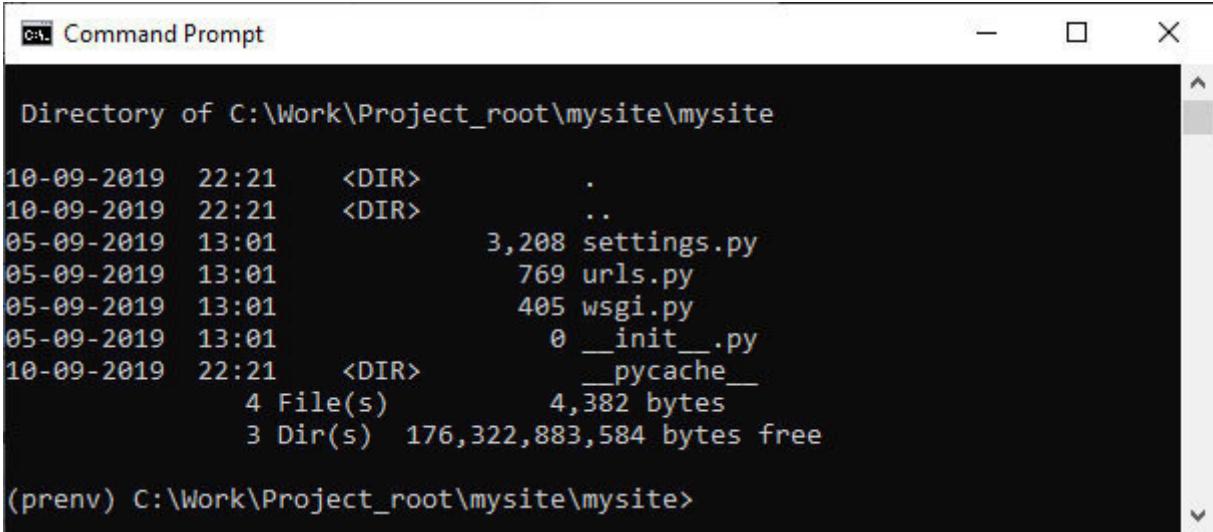
Figure 1.16: Exploring Django project files

The db.sqlite3 is the database file. By default, Sqlite3 is the database used by Django. So, you will see a database file. This file will contain all your database entries.

The manage.py is the Python file responsible for running scripts involving Django-admin and your Django project.

The mysite project folder is the internal project folder which will contain internal project files. Make sure that you do not confuse with the two mysite project folders. The Django project folder, which is inside the root folder, is the main folder that contains the internal project folder and manage.py file. The internal Django project folder will contain all the code. Both the Django project

folders have the same name that you are not supposed to change.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The title bar has standard minimize, maximize, and close buttons. The main area displays a directory listing for the path "C:\Work\Project_root\mysite\mysite". The listing includes the following files and folders:

Date	Time	Type	Name	Size
10-09-2019	22:21	<DIR>	.	
10-09-2019	22:21	<DIR>	..	
05-09-2019	13:01		settings.py	3,208 bytes
05-09-2019	13:01		urls.py	769 bytes
05-09-2019	13:01		wsgi.py	405 bytes
05-09-2019	13:01		__init__.py	0 bytes
10-09-2019	22:21	<DIR>	__pycache__	
		4 File(s)		4,382 bytes
		3 Dir(s)		176,322,883,584 bytes free

(prenv) C:\Work\Project_root\mysite\mysite>

Figure 1.17: Exploring Django project files

Now, change the directory to the internal Django project folder. Here, you will find the following files:

Here, you have the settings.py file, urls.py, wsgi.py file, __init__.py file, and the __pycache__ folder.

The settings.py file will have all the configurations and settings of your project.

The urls.py file will contain a mapping of URLs and the corresponding action to be taken when that URL is hit.

The wsgi.py file is the file that creates a web server gateway interface for your Django project.

`__init__.py` is the file that is present inside any folder with the Python code to indicate that the folder can be used as a package.

We will discuss each of these files in detail in the upcoming chapters.

Conclusion

A web framework is a tool that can be used to quickly create a web application. All the code needed for the heavy lifting is already done in a framework and all we need to do is write the code specific to your application. Virtual environments are the key to keeping the dependencies separated so that we can work on different projects at the same time.

You learnt how to create virtual environments and how to use them. You learnt how to create a Django powered project. You learnt to run the server and start your Django project. You know what files are created when you create a project in Django. You also learnt what a framework is and what to expect out of it.

In the next chapter, you will read about the Django architecture. How the Django project files solve their purpose and which file falls under which layer of the Django's MVT architecture?

Questions

What is Django?

What is the purpose of creating virtual environments?

What are the files created when Django is installed?

What is the purpose of the manage.py file?

What is the purpose of the wsgi.py file?

Create a Django project and test it by running it on the server.

CHAPTER 2

Overview of MVT Architecture

Introduction

Models-Views-Templates (MVT) is the architecture followed by Django. The other common framework follows a slightly different architecture called **Models-Views-Controller (MVC)**. The MVC is used in many other frameworks specifically those build on Java or C#. Before you start working on a Django project, you must be familiar with the architecture of the framework.

In this chapter, you will read about architecture. You will learn about Django's MVT architecture. You will also learn about the flow of the MVT architecture and how the Django's project files are layered to serve different purposes.

Structure

What is architecture?

What is MVT?

What are Models in Django's MVT?

What are Views in Django's MVT?

What are Templates in Django's MVT?

Objectives

Understanding the concept of architecture.

Understanding the Django architecture.

Understanding the purpose of different files in the Django project.

What is architecture?

Any project in the computing world has several components like a database, a presentation layer, a client (user) end, a server end, networks to connect these components, etc. If all these components are not connected in the most efficient manner, then the application will not be able to give its performance. Thus, we need a proper setup or design which shows how to connect or arrange these components such that they work at their best. Such a setup is called

Now, let's talk about web applications in particular. In a nutshell, what components do we have in a web application? A database will contain all the data, a backend, a frontend, a server to host all this, and a network to connect these components.

So, we need an architecture that can connect our database to our backend so that our code can read/write data from the database, a connection between backend and frontend so that data can be rendered (shown) on the frontend, and networks so that the flow of these communications keeps happening. This requirement is fulfilled by Django's MVT architecture.

The most common web application architecture is **Model-View-Controller (MVC)**. The **Model-View-Template (MVT)** architecture is slightly different from MVC. It does not matter if you do not have any prior understanding of MVC or the MVT architecture. You

don't need to have prior knowledge of any architecture before learning MVT from this book. We will be covering the entire MVT architecture from scratch also.

Now, before you dig deeper into Django's MVT, let's have an understanding of how a web application works. Let us take an example of a website. Go to your browser and open [Let us get](#) Let us get an overview of what happens when you hit this URL.

Firstly, your browser acting as the client sends a request to fetch the data to be displayed on the screen for the given URL. Your browser firstly checks for a cached DNS record on your computer; if it is not found, it looks for the server hosting the mentioned URL. Secondly, when the server is found, the request is received by the server. The server then evaluates the URL, validates the request received, and based on authorization it looks for data in database.⁵ Thirdly, based on the preceding computations, a response in terms of HTML code is generated and is returned to the browser.

Finally, the browser decodes the HTML and shows up the data received from the BPB server. On a broad level, this is how you will see websites on your browser.

What is MVT?

Now, you know what it takes for a website to show up on your browser. Let us see how Django's MVT architecture does that. Before moving forward, make sure that you are well aware of the components of your Django project discussed in [Chapter 1: What is Django?](#)

Let us assume that the website <https://bpbonline.com/> is built on Django. So, when you hit this URL, your browser becomes the client (user) and sends a request to the server (where code is present). The get request contains several information like the URL requested, information of the user, details of the browser, etc.

When the server receives this request by your browser, it looks for the URL in request and then tries to map that URL with the list of URLs present in the urls.py file of your project. When the requested URL is matched with any of the URLs present in the list, the corresponding views function is called from the views.py file. This function in the views.py file will be performing all the operations and then will return an HTML template from Templates as a response to the request received from the browser.

So, now you have an idea about the View, Template, and Url of MVT. 'M' here represents the Models.py file. The Models.py file is used to create tables in the database. It is a way to handle the

database queries using Python code from your Django project instead of going to any SQL editor and performing databases related tasks. So, before rendering any HTML response, the function in the views.py file connects to the database to fetch information needs to show on the page and then the HTML response is created which is rendered to the browser.

The MVT architecture flow is shown in the following screenshot:

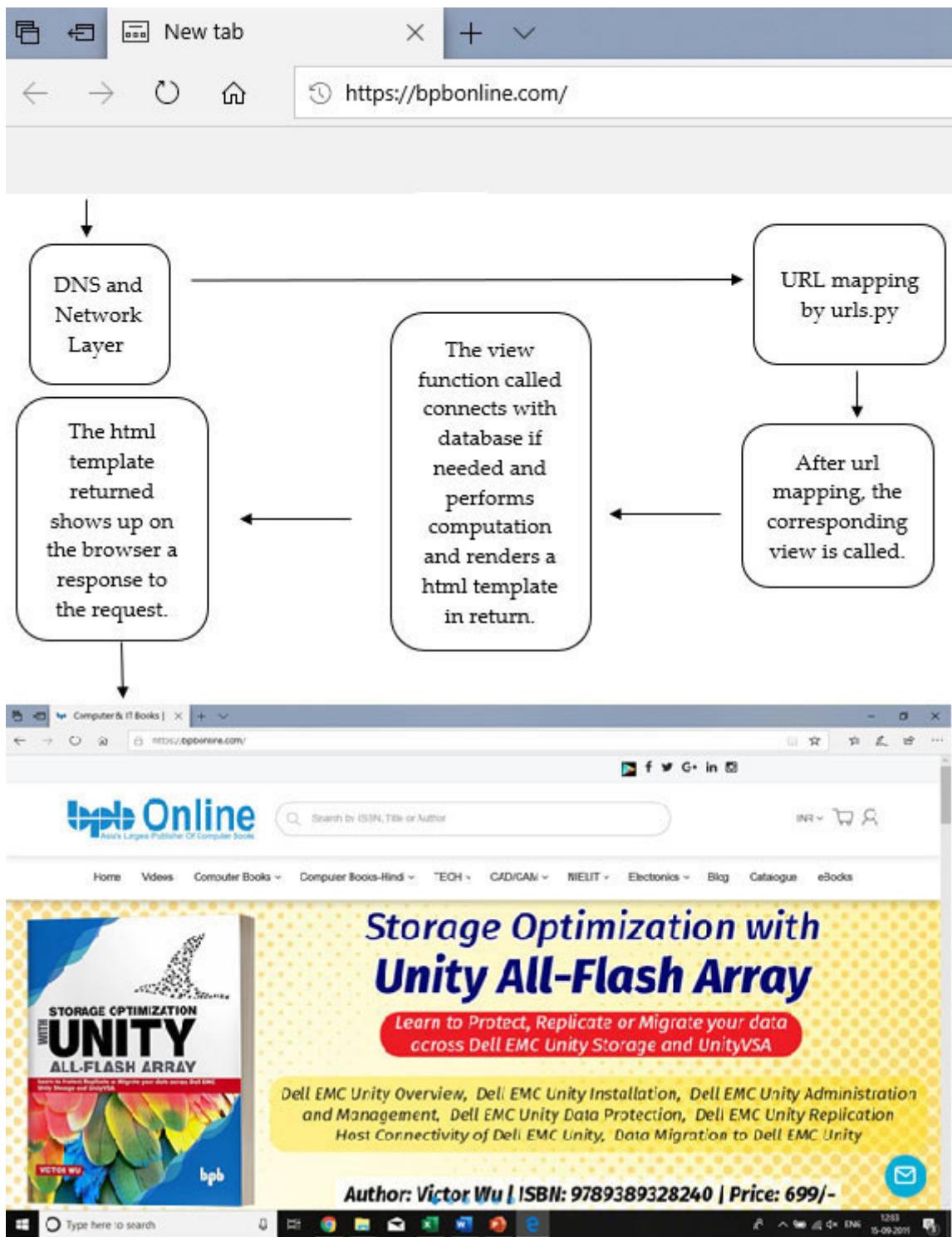


Figure 2.1: Flow of a request to the response cycle of a Django project.

So, now you have an overview of the MVT architecture. Let's dig into the components of the architecture.

What are models in Django's MVT?

Every application needs to have some way of storing data. The data could be user information, data about the content to be delivered on the website, etc. As a programmer or developer, the ease of interacting with this data programmatically is highly needed. Thus, Models come into the picture. Models are a way of describing the type of data we are going to store. The model contains classes that are representations of the database tables to be created. Each of these classes has different database tables. The classes contain attributes that represent the columns of the tables. While defining the attributes in the class, we also need to define the type of attribute and different other parameters like character size, input primary keys, etc. similar to defining any column while creating a table. The tables in the database and classes in models are the replica of each other.

Once we have our classes ready, we will create objects of these classes. When you create an object, all the mandatory attributes of the object have to be defined. This is similar to making an entry to a database table using the insert query. A new row is created with all the mandatory columns defined. Thus, a row in the database is similar to an object of a model class. So, the models provide a way to interact with database tables and rows using classes and objects. It becomes very easy to handle database operations. This was the key idea of having Models; to handle database operations in your code regardless of the programming

language. We will discuss more on this when digging down Models. For now, let's take a look at the following example.

Let's say you are building an online bookstore. You need to store data of several thousand books. You will need a database table that can store the information like name of the book, ISBN, author name, category of the book, price, edition, publisher, etc.

Let's create a database table for this requirement. The query will be somewhat like this code snippet:

```
CREATE TABLE book_info (
    book_name varchar(255),
    ISBN varchar(255),
    author varchar(255),
    category varchar(255),
    price int,
    edition varchar(255),
    publisher varchar(255)
);
```

When you run this query in an SQL editor connected to a database, it will create a table for you in database with the mentioned columns. Now, if you choose to do the same using Models, this is how you will do it:

```
class Book_info(models.Model):
    book_name = models.CharField(max_length=150, null=False)
    ISBN = models.CharField(max_length=150, null=False)
```

```
author = models.CharField(max_length=50, null=False)
category = models.CharField(max_length=700, null=True)
price = models.IntegerField(max_length=150, null=True)
edition = models.CharField(max_length=150, null=True)
publisher = models.CharField(max_length=150, null=True)
```

Adding this class to the `models.py` file of your project will create a similar table in database. Then, the objects of this class will be just like inserting rows in a database table. This concept is also called **Object Relational Mapping (ORM)**. ORM is a programming paradigm which interacts with the database using object-oriented programming concepts of classes and objects you saw earlier.

ORM is implemented using libraries. In Python, we have Django's inbuilt ORM library and SQLAlchemy library. This concept of Models is based on ORM which is inbuilt in Django. So, you don't have to worry about learning any library dedicated to ORM. Another advantage of Django!

What are Views in Django's MVT?

The views are the controllers in Django's MVT architecture. This is where all the logic is implemented and all the computations are done. This is a layer that works on the response to be rendered. The urls.py file maps the URL requested to the list of URLs present in the file and calls a view function corresponding to the URL. The view functions are defined in the views.py file. Here are sample urls.py and views.py files:

The screenshot shows two Python files in a code editor: `urls.py` and `views.py`. The `urls.py` file defines URL patterns, and the `views.py` file contains view functions. Colored arrows indicate dependencies between them.

urls.py:

```
1 from django.contrib import admin
2 from django.urls import path, include
3 from blog import views

4

5 urlpatterns = [
6     path('', views.home, name='homepage'),
7     path('index', views.home, name='homepage'),
8     path('elements', views.elements, name='elements'),
9     path('generic', views.generic, name='generic'),
10    path('admin/', admin.site.urls),
11]
12
13
14]
```

views.py:

```
1 from django.shortcuts import render, get_object_or_404
2 from .models import Post, Comment
3 from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

4

5 def home(request):
6     return render(request, "index.html")
7

8

9 def generic(request):
10    return render(request, "generic.html")
11

12

13 def elements(request):
14    return render(request, "elements.html")
```

Figure 2.2: Co-relation between Django `urls.py` and `views.py`

In the preceding screenshot, you can see a snapshot of two files – `views.py` and `urls.py`. You can see a connection between the two files. Let's first take a look at the `urls.py` file. Here, you can see a few import statements, ignore them for now. Then, you see a list named `urlpatterns`. This is a list of `path` function calls with different arguments.

The first argument is a string also considered as an expected URL. The concept of regular expressions also known as regex is used while defining the first argument or expected URL. The second argument of the path function call is the view function to be called if the first argument or the expected URL matches the URL from the request. Follow the curved lines in the screenshot. It shows the connection between the views.py and If the URL matches with the first element of then the second argument of that element (the path function call) points to a function from the views.py file to be called. In this case, if the first URL matches the URL in the request, then as mentioned in the second argument of the first path function call views.home, the home function from the views.py file will be called.

Now, focus on the views.py file. The home view function has been called and has received the request as an argument. Now, the job of the home view function is to execute the logic mentioned inside and then at the end, returns a template in response to the request received. In this case, the home function has no logic to implement this; it will execute the return statement and render the template named index.html to the request. This HTML template will be received by the browser and a webpage will appear on the screen. You must have a better idea of the flow of requests and responses now.

Now, consider a scenario where the URL in the request does not match with the first path function call in the urlpatterns. Then, the url-mapper will try to map the URL in the second path function call, and so on. If it matches, then the corresponding view function will be called, else the matching will continue with

the third path function call, and so on. If none of the path function calls matches with the URL in request, then the inbuilt 404 error (page not found) template is rendered by Django.

What are Templates in Django's MVT?

The HTML response which is finally rendered to the client is the Template layer. So, we can say that the Django output is an HTML Template. So, the next question that comes to mind is what the content inside these templates is and how is that content controlled or manipulated before rendering. Let's dig in.

Through Templates, Django provides you a way to write your frontend HTML code and manipulate the content inside that HTML programmatically using Python code. So, basically, templates are a mixture of Python and HTML code written in a special way using tags. You have tags of both Python and HTML code. This special way of writing Python and HTML together is called There are several template engines available in which you will write your templates like Jinja2, Mako, or the default Django Templating, etc.

Let's see how a sample template looks like:

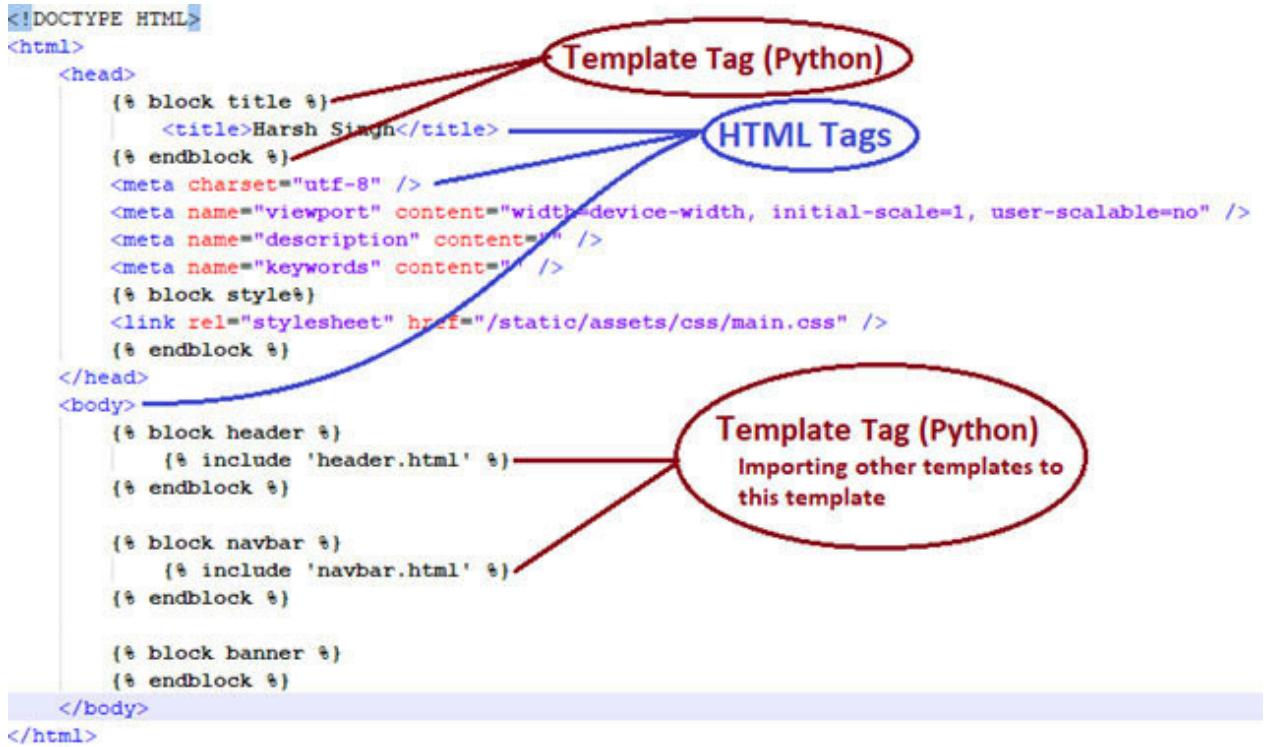


Figure 2.3: A sample Django template

Here, you see an HTML code with HTML tags like head, title, etc. and Python template tags like block, etc. This is how a sample template in Django looks like. HTML and Python code together. Templating empowers you to create HTML dynamically. You can implement for loops using Python code to dynamically create HTML tags. You can dynamically generate href tags based on the requirement. You can import the common HTML like header, etc. to all other HTML pages of your website as shown here. It will save you from repeating yourself. You can implement conditional operation and choose when to implement a certain section of HTML code in your template.

So, templating simply gives you a way to handle the static frontend dynamically through Python code. We will discuss templates in more detail in the upcoming chapters. If you wish to know more

about templates, feel free to browse the official Django template documentation at

Conclusion

In this chapter, you learned about Django's MVT architecture. The goal here was to give a clearer picture of the way components are arranged in a Django project, the way different components communicate with each other, and the roles these components must play. You learned how models help us interact with the database using Python code. You learned how the response is controlled by the views functions and how templating makes it easy to dynamically handle static HTML code. You also learned how to write basic models, views, url-patterns, etc. You also learned how to structure your Django project.

In the next chapter, you learned about another import component of your Django project – settings.py file. You created a Django project in the previous chapter. Keep that project handy. You will be working more on that.

Questions

What is Django's MVT architecture?

How is the database handled in the MVT architecture?

Where is the logic of the project written?

How does Django map the request from the browser to the logic?

How is the frontend handled in Django projects?

CHAPTER 3

Understanding Django Settings

Introduction

Any framework is like a tool that has programs to do all the heavy lifting and when an application is developed using the framework, all a programmer needs to do is to write the code relevant to the particular application. The settings of the Django framework is the configurations that you need to do so that the framework knows which resources to look for and where. Thus, it becomes very important to understand these configurations.

In this chapter, you will read about the setting and configuration of your Django project. The focus will be on the contents of the `settings.py` file. You need to know about the database, middleware, internal and external application integrations, root folder locations, frontend files locations, content information, etc. All these configurations or settings need to be set somewhere. In Django, this entire configuration happens in the `settings.py` file.

Structure

Exploring some common settings

Core settings

Authentication

Static files

Objectives

Understanding the common settings.

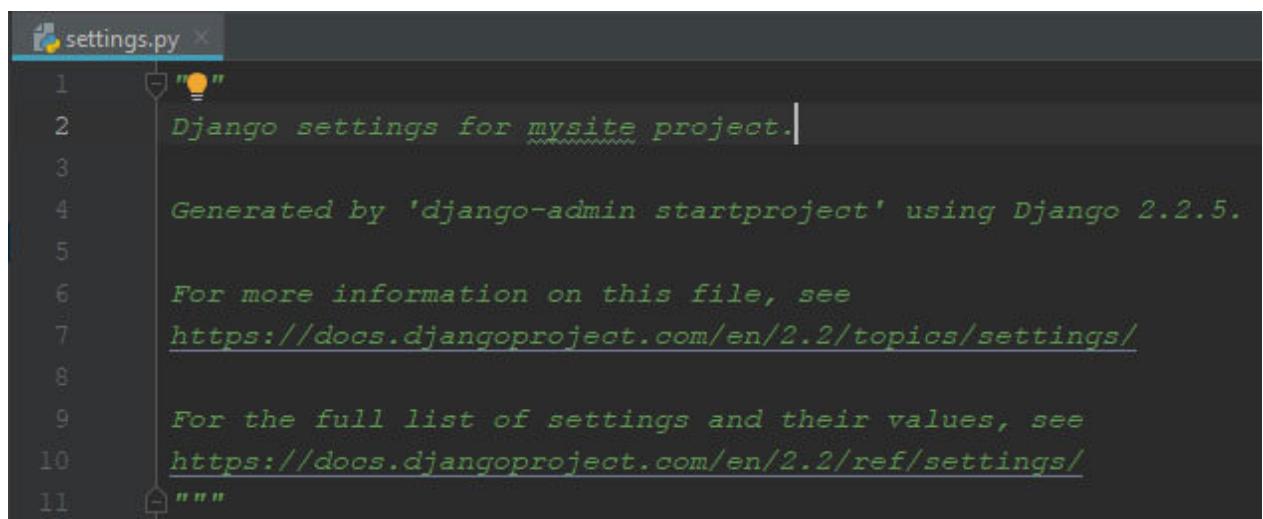
Understanding the impact of settings on the project.

Understanding some external configurations.

Exploring some common settings

In this section, we will be simply exploring the setting.py file of the project you created in the previous chapter. You will see what are the default settings. In the upcoming sections, you will read about modifying and adding other settings as per your requirement. Now, just open your settings.py file, and you will see the following arguments and objects:

The comments at the top in the following screenshot: The comments at the top are auto-generated when a project is started. It states the name of the project and the version of Django. It mentions two URLs to the official documentation which can be referred to while working with a Django project:



A screenshot of a code editor showing the contents of a file named "settings.py". The file contains the following text:

```
1 """  
2 Django settings for mysite project.  
3  
4 Generated by 'django-admin startproject' using Django 2.2.5.  
5  
6 For more information on this file, see  
7 https://docs.djangoproject.com/en/2.2/topics/settings/  
8  
9 For the full list of settings and their values, see  
10 https://docs.djangoproject.com/en/2.2/ref/settings/  
11 """
```

Figure 3.1: settings.py file

The import os statement simply imports the os library.

BASE_DIR: This variable sets the base directory of the project. By default, it will be the folder where your project lies.

SECRET_KEY: This is the secret key specific to your Django project. It is set to a random value by Django when you start a project. This key is needed for Django to work properly. It also aids in password reset tokens, messages while using cookie storage, and handling sessions.

DEBUG: This is a very useful setting for a developer. The default value is True. If the value is set to True, if there is an error in your Django Project, then you will be able to see a very detailed error message on the browser screen. Please note that it is very important that you set this to False on your production servers. Because if the error details show up to the world, it makes your project very vulnerable.

ALLOWED_HOSTS: This is a list of hosts/servers or domain names on which your Django project will run. This is a security for stopping your Django project to serve any host header attacker.

This list can contain a website name like www.123qer.com or some IP address or simply localhost for running local instances.

INSTALLED_APPS: This is a list of inbuilt applications or external apps enabled in your Django project. It contains the list of the apps installed or integrated into your Django project. When you

create different apps in your project, those apps need to be appended to this list. If you are using any app of any project, that has to be appended to the list. There are a few apps enabled by default. Let's take a look at the uses of those apps:

This is the Django admin app. It's like a gift from Django. You get a very powerful and secure admin portal for your project inbuilt. You can use this app to manage users, passwords, and data corresponding to your project.

You can access this in your current project at `/admin`. You will need super user credentials to log in to this.

Auth: This app is used to register users and authenticate them. It uses the inbuilt User model and its methods to perform user registration, login, and password reset functionalities, etc.

Contenttypes: This inbuilt application looks for the models installed in your Django project. It is a model that stores information about the models in your Django project. Each instance of the Contenttype model is a representation of a model installed in your Django project. An instance of Contenttypes contains two fields: `app_label` (app name in which the model is defined) and `model` (name of the model class). We do not have any app in our project as of now.

Session: This app is used to enable the session framework of Django. It is used to store and retrieve data on a per user basis.

Messages: The Django framework provides a setup for a cookie-and session-based messaging framework for all kinds of users. You can display messages of different levels like info, error, or warning depending on the kind of user.

Staticfiles: The Staticfiles app is responsible for gathering all the static files from all the applications that are specified in the installed apps and from any other locations mentioned in a single location so that it can be rendered readily.

MIDDLEWARE: A middleware is a bridge between different components of software. In Django, middleware is a low-level plugin system for globally altering Django's input or output. Each middleware component is responsible for performing a specific task. Let's check out the inbuilt middleware available by default:

This is implemented through the `SecurityMiddleware` class responsible for security through the request/response cycle. It offers several configurations that can be modified individually as per requirement.

SessionMiddleware: This is implemented through the `SessionMiddleware` class. It is responsible for handling user sessions of your Django application. You can configure this middleware the way you want to use your sessions. If you don't want to use sessions in your application, you can remove it from the list.

CommonMiddleware: This is implemented through the CommonMiddleware class. It is responsible for setting some of the key configurations. It is generally used by experienced developers to handle redirects, user agent permissions, etc.

CsrfViewMiddleware: CSRF is cross-site request forgeries. This middleware is responsible to protect your application from such threats.

AuthenticationMiddleware: This is implemented through the AuthenticationMiddleware class and is responsible to track the activities of the logged-in user. It validates if the user logged in has the required permissions to perform any activity.

MessageMiddleware: This is implemented through the MessageMiddleware class and it helps in setting up the session's storage and cookie management.

It is responsible for safety against threats like clickjacking.

ROOT_URLCONF: This is the module that is to be used for URL matching. Generally, it is set to

TEMPLATES: Here, you set the directory of your project templates which will be rendered in response. It is set in such a way that the path to your templates can be easily defined in the views functions.

Server Gateway Interface (WSGI) is a protocol followed between a framework and the web server. This is responsible for how the server will send a request and how the framework will respond to it. In short, this is the interface Django uses to communicate with servers.

DATABASES: Here, you can configure your databases. Django supports a variety of databases. You need to configure it here.

AUTH_PASSWORD_VALIDATOR: This is responsible for validating passwords. Formats like these should not be similar to the user name and should have a minimum length, etc.

LANGUAGE_CODE: This is used to set the language for the frontend.

TIME_ZONE: This is used to set the timezone for the application.

STATIC_URL: This is the path for your static files inside your app. You may choose to keep the static files of your project segregated app-wise or you can keep them all in one place. Just configure it accordingly.

STATIC_ROOT: This is the path to the static folder.

You may skip the rest of this chapter for now and come back later after you have been through the mini project.

Core settings

There are several settings you can do in a Django project and customize it to the way it suits the need of your application. Some of them have been described as follows:

It is a dictionary used to override the absolute URLs which may be defined in the models.py file. Example:

```
ABSOLUTE_URL_OVERRIDES = [
    'student.studentdetails': lambda o: "/studentdetails/%s/" % o.student_id,
```

Figure 3.2: settings.py file

This is a list of tuples with two elements. The first element is the name and the second element is email ID. This list is used by loggers to send emails when there is any error on the production site or if there is any error when the Debugging is turned off.

This is a list of domains where the application can be hosted.

If this is set to true, a / is appended at the end of the URL, if the URL does not match with any of the patterns defined in urls.py file.

DATA_UPLOAD_MAX_MEMORY_SIZE: The maximum size of data in bytes which can be processed in a request. The default is

The maximum number of fields a request can have. The default is

This is the format in which the date field will be used. Default – N j,

You would want to configure the logging of your production environments when the DEBUG is false to investigate the error. This can be done here.

There are several other core configurations that you can find in the official Django documentation.

Authentication

Django provides an inbuilt authentication system. You can use it for authentication and authorization. Authentication lets users log in to the application and authorization helps configure and enforce the roles and permissions of users. It makes it very easy for developers to focus on the real problems of the application and not think about developing an authentication and authorization system. It also has an inbuilt feature of email ID validation. You can also configure your application to use social authentication like login Google, Facebook, etc.

Django provides a detailed documentation for using and customizing the Authentication model. You will read about this module later while building the project. You can find the documentation at [You learn how to set up the authentication step-by-step later while setting up the project.](#)

Static files

The Django project uses static files like HTML files, CSS, JavaScript files, etc. These can be placed at the default location and you don't need to worry about configuring settings for these. You may want to store your static files at different locations based on your requirement. In such circumstances, you need to configure the static_files settings:

STATIC_ROOT: Default: None

Django runs a function collectstatic which stores the static files at this location if STATIC_ROOT is set to a path. The following screenshot shows the execution of the collectstatic command:

```
(penv) C:\Work\Project_root_old\mysite>python manage.py collectstatic
121 static files copied to 'C:\Work\Project_root_old\mysite\static'.
```

Figure 3.3: STATIC_ROOT-collectstatic example.

Here, the STATIC_ROOT = static file is placed at the location.

STATIC_URL: Default: None

When your Django power pages loads, the static files are loaded on the browser. They need a URL to be loaded. This sets their

URL. It may be just the domain name appended with /static/ or a completely different URL location may be of another website.

STATICFILES_DIRS: Default:[]

Here, you can set additional locations of static files that the findstatic and collectstatic functions will use to fetch the static files used.

STATICFILES_FINDERS:

Default: ['django.contrib.staticfiles.finders.FileSystemFinder', 'django.contrib.staticfiles.finders.AppDirectoriesFinder',]

These are the engines used to fetch the static files.

STATICFILES_STORAGE:

Default: 'django.contrib.staticfiles.storage.StaticFilesStorage'

This is the storage engine to be used while storing the static files found and collected.

You do not need to worry about most of these settings. These are to change under very specific circumstances.

Conclusion

In this chapter, you read about different common settings like DEBUG, ALLOWED_HOSTS, BASE_DIRS, etc. We discussed the database configurations, static files configuration, templates configuration, etc. Now, you would be able to do configurational changes to your project.

You may have an idea of the usage of these setting options. Although the motive here is to give you a brief idea of how customizable Django is. There are a lot of such settings and since Django is open source, you can customize it as much as you like. To keep up with industry standards, it is suggested that you do not alter the framework files and customize them only using the settings.py file. You will get to work a lot on this file in the upcoming chapters while configuring your projects.

In the next chapter, you will learn about Django's built-in admin app. You will work on the project created in the previous chapter. You will create a super user and use it to log in to the admin app and explore it.

Questions

What are the common settings of a Django project?

What is the use of

What is the significance of

What is the use of the ALLOWED_HOSTS setting?

How do you set the template directory?

How does Django handle the database configuration?

What are the different dataset engines supported by Django?

CHAPTER 4

The Django Admin Utility

Introduction

Every web application has different kinds of users. If we have a shopping website, then it may have users like customers, sellers, delivery guys, admin, and maybe more. Regardless of the type of application, every application has an admin portal from where the administrators can control the application. In most of the frameworks, programmers need to design and develop their admin portal. In Django, this portal comes with the framework. You can use it straight away. The admin portal grows along with your application but you must know how to configure and use it.

In this chapter, you will be making an app in your Django project which will take details of students in a school and feed them in a database. We will start where we left our Django project. Keep the project handy. You will learn about the admin page and see how it looks now, and then you will create an app and see how the admin page looks like then. You will get to see a working ORM and its flow.

Structure

Admin page of a Django project

Creating an app

Editing settings.py, and admin.py

Adding data to your database

Editing data in your database

Objectives

Understanding the login admin page of a Django project.

Understanding how to create an app.

Understanding the working of the built-in admin app.

Admin page of your Django project

Python provides a built-in interface for admin functionalities. It is more like an app that is already created and set in the Django project for developers to use. If you check the list of installed apps in the settings.py file of your project, you will find the admin app there in the

Let's go back to the project you created and execute the python manage.py runserver command. You should be able to see the following screen:

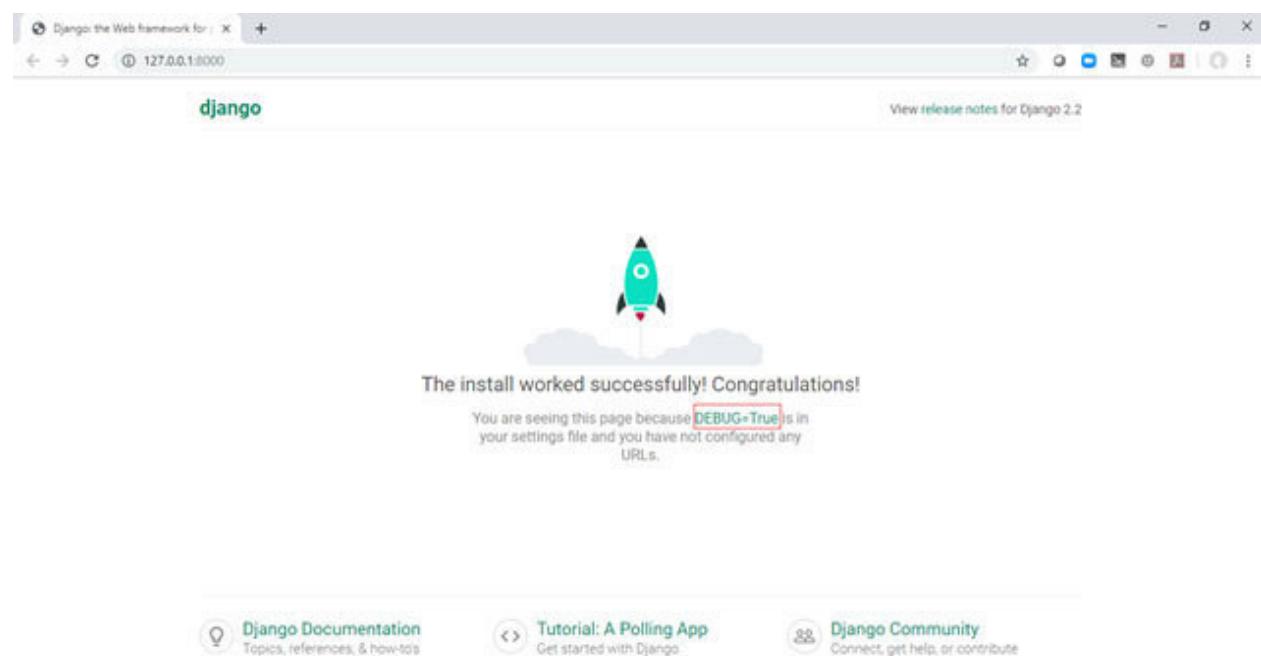


Figure 4.1: First Django powered page

Now, if you see this screen well and good. If you don't, just go back to the previous chapter where we created a Django project and follow the steps thoroughly. Next, in your browser, edit the URL to This will open the Django admin login page for you. Now, we need to have super user credentials to log in to the admin page.

Let us create one. Shut down you server + C in the terminal) and run the following commands in the sequence as shown in the following screenshot:

python manage.py makemigrations: This command checks for any changes in the models.py file. Since we have not made any changes in the files yet, you will see the message No changes

```
(penv) C:\Work\Project_root\mysite>python manage.py makemigrations
No changes detected
```

Figure 4.2: executing the makemigrations command

python manage.py migrate: This command implements the model classes from everywhere in the project and creates or modifies the database accordingly:

```
(penv) C:\Work\Project_root\mysite>python 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 sessions.0001_initial... OK
```

Figure 4.3: Executing migrate command

`python manage.py createsuperuser`: This is a command which creates the super user for your project. This user will be the ultimate admin of your project with access to perform any task. Once you have a super user, you can create other users with limited access as per need.

When you execute this command, you will be prompted to provide a user name (here: email id, and a password. Once you have created a super user, you can log in to the admin with the same credentials:

```
(penv) C:\Work\Project_root\mysite>python manage.py createsuperuser
Username (leave blank to use 'awani'): bwwg
Email address: awanish.infarna@gmail.com
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.

(penv) C:\Work\Project_root\mysite>
```

Figure 4.4: Creating superuser

Now, running `python manage.py runserver` command starts the server. Then, add `/admin` to the end of the URL in your browser. You will see the login page again:

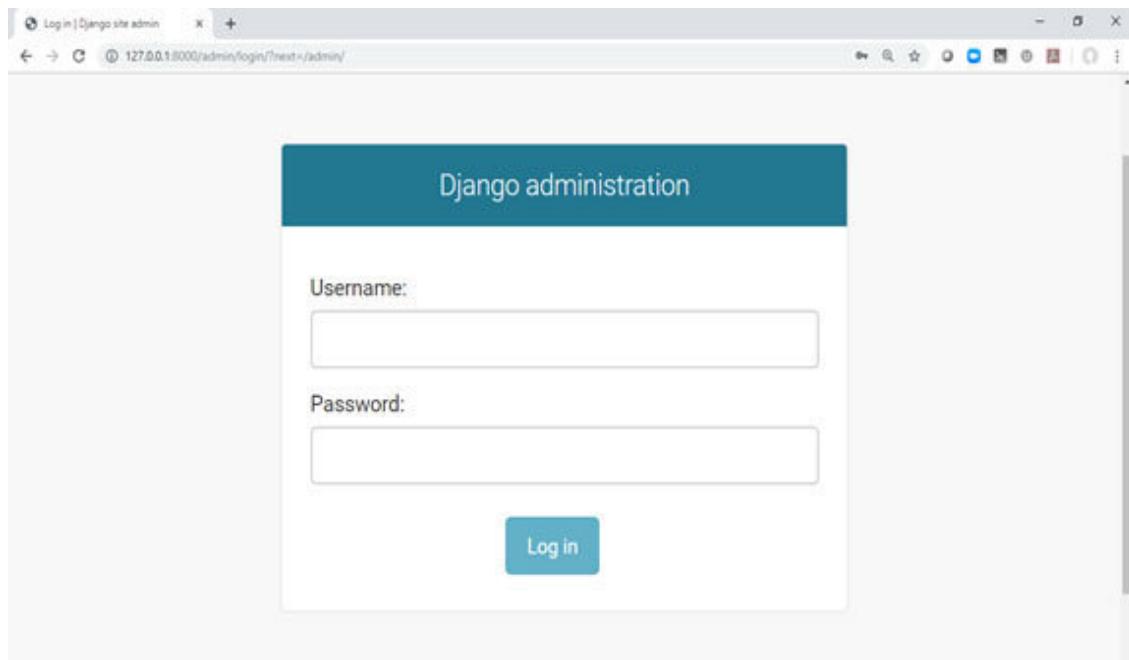


Figure 4.5: Admin page of your project

Enter your credentials and hit You will be logged in and the following page will appear on the screen:

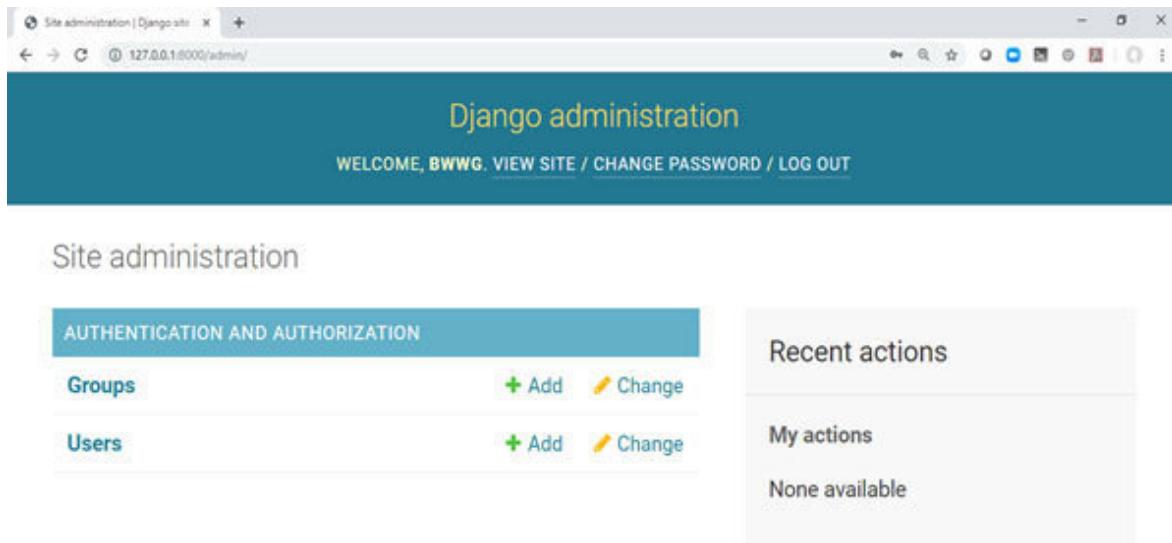


Figure 4.6: Admin app of Django

This is the site administration page. Here, you see a section **Authentication and Authorization**. In this section, you can manage users and user groups. The two items Groups and Users showing up here are the two model classes of the admin app. You can create a **user** in the Users model, and it will create an object of the **user** model and then that object will be inserted in the database as a new row. Let's add some users and see how the admin app behaves. On the preceding screen, click on the **Add** link corresponding to the **user** row. The following page will show up:

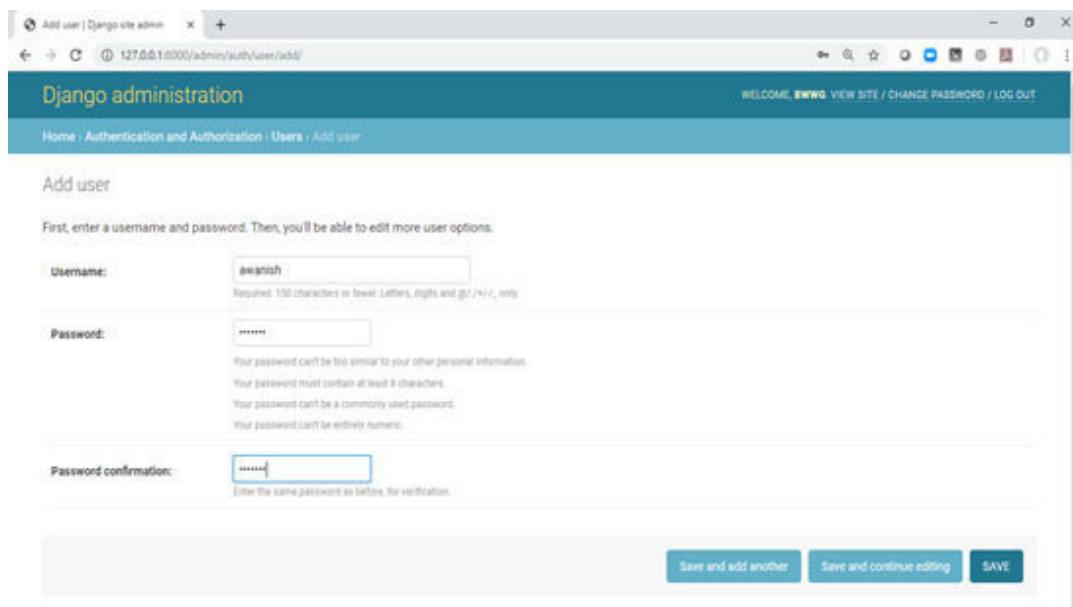


Figure 4.7: Adding users using the admin app

Fill in the details and click on Then, you will see a screen where you can fill all the details for the user you just created:

The user 'awanish' was added successfully. You may edit it again below.

Change user

Username: Required: 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: algorithm: pbkdf2_sha256 iterations: 150000 salt: bVU1Lh***** hash: KmB87m
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using [this form](#).

Personal info

First name:

Last name:

Email address:

Permissions

Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status
Designates whether the user can log into this admin site.

Superuser status
Designates that this user has all permissions without explicitly assigning them.

Groups:

Figure 4.8: Editing user details

You can set the permissions, access limits, etc. Once done, you can go back to the site administration page and click on Now, you can see the two users present here. The superuser and the one you added just now:

The screenshot shows the Django admin 'User' list page. At the top, there's a success message: 'The user "awanish" was changed successfully.' Below it, the title 'Select user to change' is followed by a search bar and a 'Search' button. A 'FILTER' sidebar on the right includes sections for 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active'. The main table lists two users: 'awanish' (Awanish Ranjan, staff status red circle) and 'bwwg' (awanish.infarna@gmail.com, staff status green circle). The table has columns for USERNAME, EMAIL ADDRESS, FIRST NAME, LAST NAME, and STAFF STATUS.

Action	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	awanish	awanish.ranjan5@gmail.com	Awanish	Ranjan	●
<input type="checkbox"/>	bwwg	awanish.infarna@gmail.com			●

Figure 4.9: User list in the admin app

To conclude, the two users showing up here are the two entries in the Users table corresponding to the **User** model class. Thus, the models of any new app you create can be registered in the admin app, and once registered here, they will show up here and they can be manipulated the same way. This was a brief overview of the admin app. You will read more about it in the upcoming chapters.

Creating an app

By now, you must have understood the difference between a Django project and an app. A Django project can consist of several apps inside it. In a broader sense, a Django project is a product as a whole and it uses small apps to solve smaller dedicated problems.

There are two ways of creating an app:

Using the Django-admin tool

Using the manage.py script

So, go to your terminal and shut down the server. Then, execute the `python manage.py startapp student` command:

```
(penv) C:\Work\Project_root\mysite>python manage.py startapp student
(penv) C:\Work\Project_root\mysite>
```

Figure 4.10: Creating an app in your project

This will create an app named `student` inside your Django project. You can see the app folder inside the project folder. Check out the folder structure as shown in the following screenshot. It is important to be aware of the folder locations:

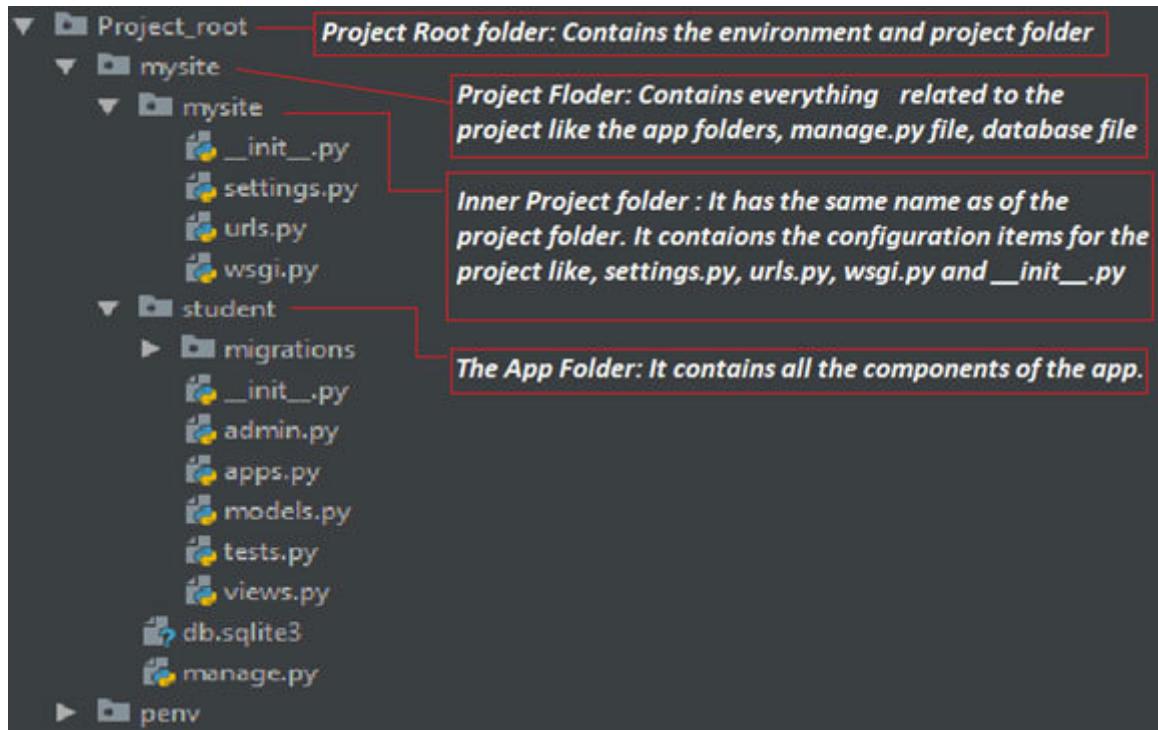


Figure 4.11: Exploring app and project folders

So, you're here now. Let's dig into the components:

admin.py: This is the file where you register the model classes of your app so that they can appear in the admin app.

apps.py: Django has a set of configurations for apps by default. If you want to change any, that can be done here. Although it is recommended that you do not make any changes unless you are fully aware of the impact.

models.py: This is the file where you define your model classes.

test.py: In this file, you define your test cases for testing.

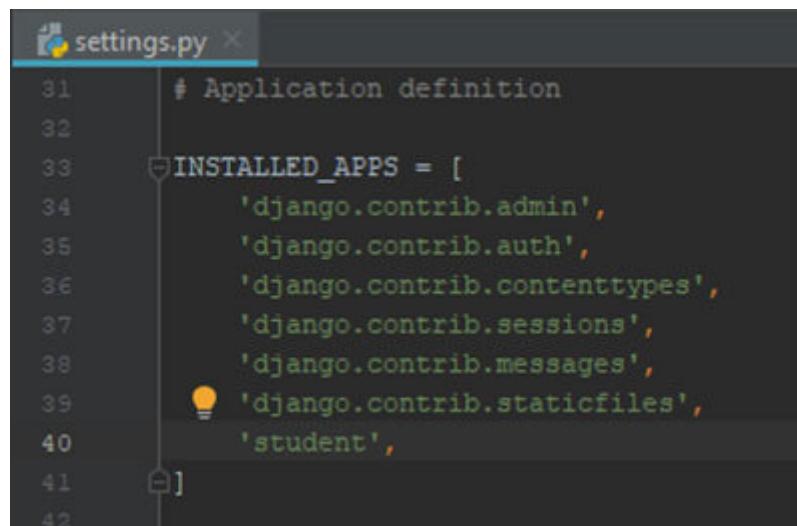
views.py: This is where your logic is implemented and a response is generated.

migrations: When you create a class in the model and want them to be captured in the database, you need to first capture the class you created or the changes you made in that class by running the makemigrations command. When you run this command, Django creates a file in the migrations folder tracking the changes you made. This helps if you wish to ever rollback any migration.

So, you have successfully created your app. In the next section, you will do some coding to make your app do something.

[Editing models.py, settings.py, and admin.py](#)

Once you have created your app, you need to tell Django that you have created an app so that Django can implement its features on that app. To do this, open settings.py in your project and add your app name in the list of INSTALLED_APPS as shown in the following screenshot:



```
# Application definition

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

Figure 4.12: Installed apps in the settings.py file

Open the models.py file in the app folder. Here, you can write your model classes. Here, you will create a class named This class will have attributes like student name, parent's name, student number, date of birth, etc. This class will inherit the base class of model Note that every model class you define should inherit the base models.Model class. Now, open your models.py file in the app folder and write the following code:

```
models.py
1  from django.db import models
2  from django.urls import reverse
3
4
5  class StudentDetail(models.Model):
6      gender_choices = [('M', 'Male'), ('F', 'Female')]
7      gender = models.CharField(choices=gender_choices, max_length=1,
8                                  default=None, null=True)
9      first_name = models.CharField(max_length=250)
10     last_name = models.CharField(max_length=250)
11     student_id = models.SlugField(max_length=250, unique=True)
12     fathers_name = models.CharField(max_length=250)
13     mothers_name = models.CharField(max_length=250)
14     dob = models.DateTimeField()
15     objects = models.Manager() # The default manager.
16     std_choices = [('1', 'I'), ('2', 'II'), ('3', 'III'), ('4', 'IV'),
17                     ('5', 'V'), ('6', 'VI'), ('7', 'VII'),
18                     ('8', 'VIII'), ('9', 'IX'), ('10', 'X')]
19     std = models.CharField(choices=std_choices, max_length=1, default=None, null=True)
20
21     class Meta:
22         verbose_name_plural = "Student Details"
23
24     def get_absolute_url(self):
25         return reverse('student:viewStudentDetails', args=[self.student_id])
```

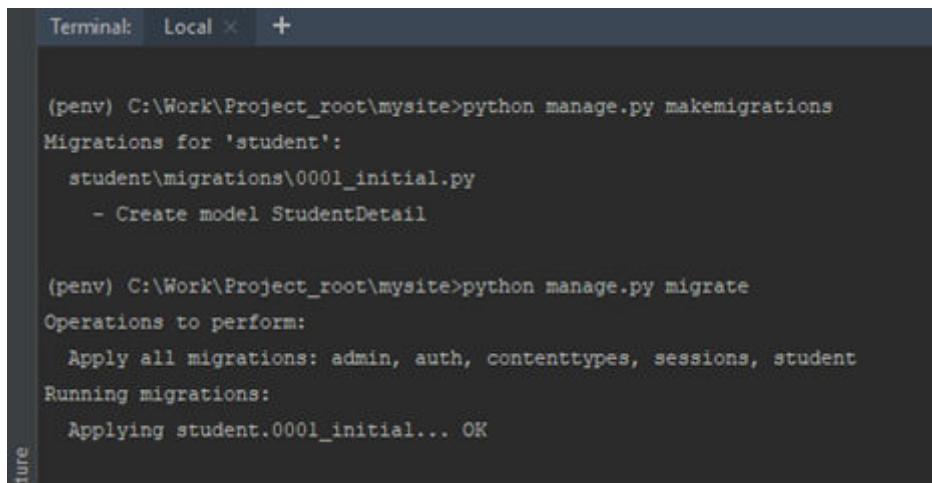
Figure 4.13: Model class in the models.py file

Once completed, now you need to register your model to the Django admin. So, open the admin.py file in your app folder and write the following code in it:

```
admin.py
1  from django.contrib import admin
2  from .models import StudentDetail
3
4  # Register your models here.
5  admin.site.register(StudentDetail)
```

Figure 4.14: Registering model to admin

This will register your model to the Django admin. Now, you need to tell your ORM to create a table in the database corresponding to the class you define. To do this, go to your terminal and activate the virtual env and change the directory to the project folder (the folder which contains the manage.py file). Then, execute the two commands `python manage.py makemigrations` and `python manage.py migrate` respectively:



```
Terminal: Local × +  
  
(penv) C:\Work\Project_root\mysite>python manage.py makemigrations  
Migrations for 'student':  
    student\migrations\0001_initial.py  
        - Create model StudentDetail  
  
(penv) C:\Work\Project_root\mysite>python manage.py migrate  
Operations to perform:  
    Apply all migrations: admin, auth, contenttypes, sessions, student  
Running migrations:  
    Applying student.0001_initial... OK
```

Figure 4.15: Making migrations

Now, your migrations have been applied, thus a table has been created in the database corresponding to the model you defined.

[Adding data to your database](#)

After executing all the preceding steps successfully, you can start your server again manage.py and log in to the admin page again. Here, you will see your app too as shown in the following screenshot:

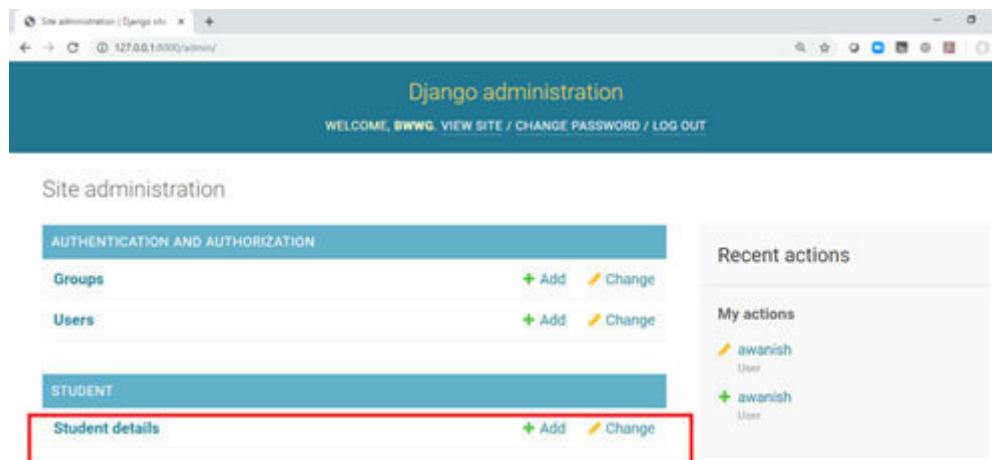


Figure 4.16: Checking the model in the admin app.

Here is an interesting observation. Nowhere in the code have we used the text Student. The name of the model class is StudentDetail and the app's name is student, but still, Django showed the model name in UI as Student. This is one of many smart features of Django. You can also control what the name of your model classes are displayed in different classes by subclasses in the model classes. You will read about it in the upcoming chapters. For now, just click on your model class on the admin page to see what you have:

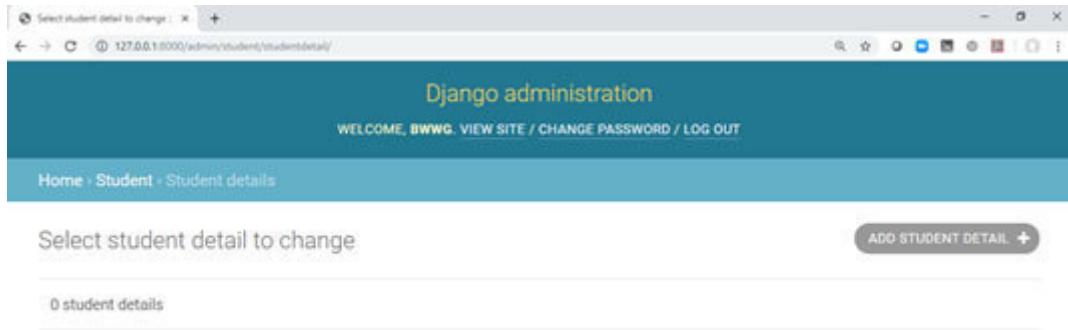


Figure 4.17: Model in the admin app

Here, you have an option to add the student detail. Another smart feature of the Django!. Click on **ADD STUDENT** You will see a form with fields you need to set in your model class definition. Fill the form and click on

A screenshot of a web browser displaying the "Add student detail" form within the Django administration interface. The title bar shows the URL as 127.0.0.1:8000/admin/student/studendetail/add/. The main header says "Django administration" and "WELCOME, BWWG. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below the header, the breadcrumb navigation shows "Home > Student > Student details > Add student detail". The form has fields for "First name" (Nikhil), "Last name" (Singh), "Student id" (0101), "Fathers name" (Raju Singh), "Mothers name" (Rita Singh), "DOB" (Date: 2009-05-05, Time: 00:00:00), and "Std" (dropdown menu with 'V'). At the bottom, there are three buttons: "Save and add another", "Save and continue editing", and a large blue "SAVE" button.

Figure 4.18: Adding data to model using the admin app

Similarly, add few more student details and then when you open your model class page, you can see a list of the objects of your model class:

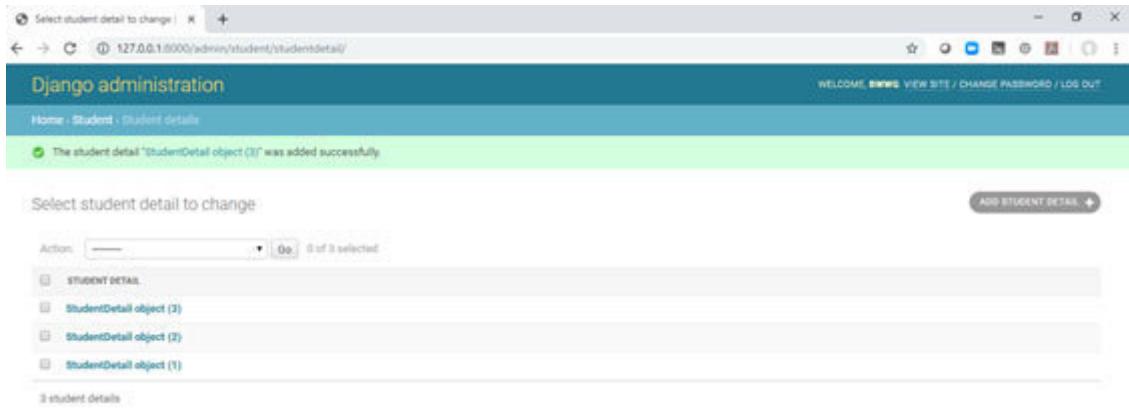


Figure 4.19: Checking objects of the model class in the admin app

This is on the way of adding data to your database. You can create as many classes as you want in `models.py` and you can control them from here itself. For example, fee payment details of the students, marks details, results, details of teachers, etc.

Editing data in your database

Since you have data in your database, let's try and see how you can edit it. Click on your model class and the list of its objects will appear as shown in the following screenshot:

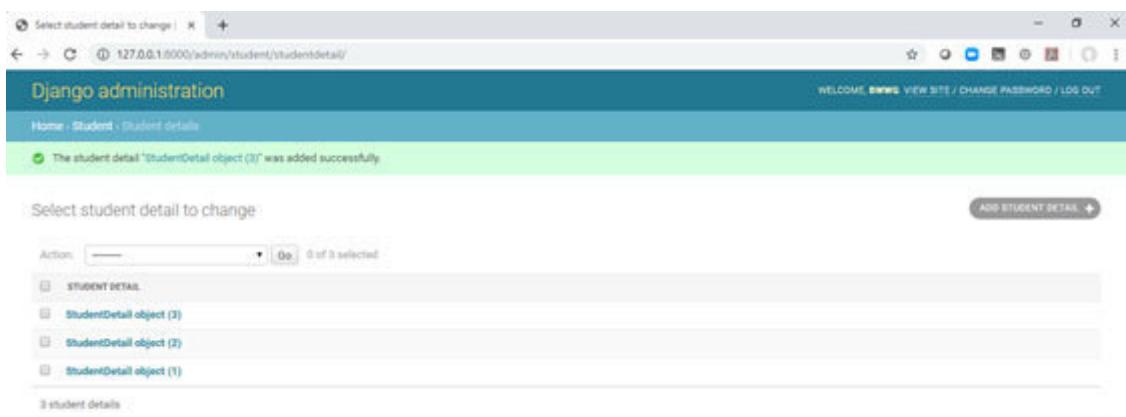


Figure 4.20: Editing data of models using admin

Now, click on any of the objects and a form will pop up on the screen:

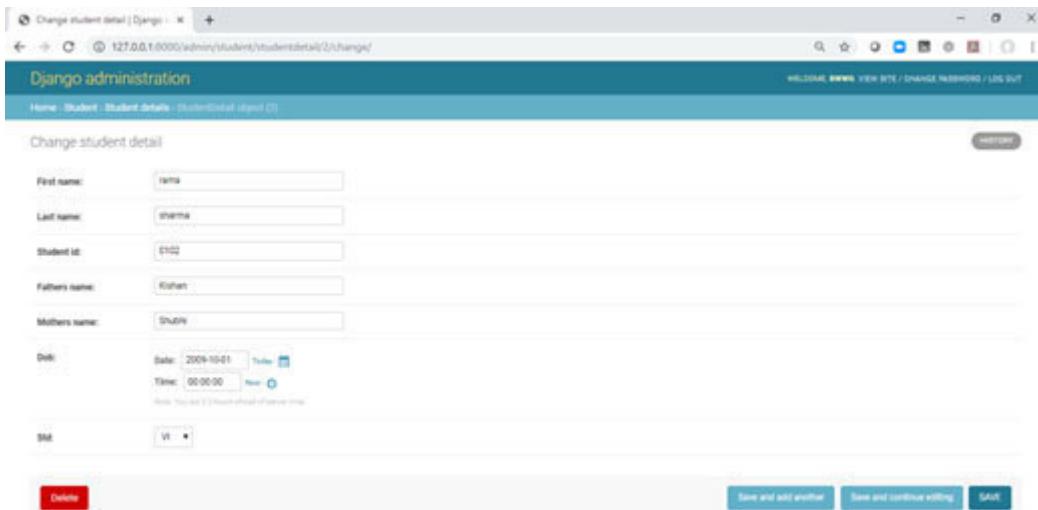


Figure 4.21: Updating data

You can edit any of the fields and click on The details will be updated. Now, open any object from the list and try to edit its Student id field. Fill it with some value that is already used by any other student and click on You will see an error:

Please correct the error below.

First name:	rama
Last name:	sharma
Student detail with this Student id already exists.	
Student id:	0101
Fathers name:	Kishan

Figure 4.22: Primary key validation check

This is because you have set unique = True in your class definition This ensures that no duplicate value is accepted for this

field.

Conclusion

In this chapter, you created an app and used the admin app of Django to play with the database of your app. You saw how to register your app and models to the Django admin and how to use the inbuilt Django Admin app to manipulate with model databases. In the next chapter, you will learn more about another way of manipulating the database. Now, you know how to create a project, create an app in it, create a superuser, and log in to the admin app. You know how to add data to the model and manipulate it using the admin app.

In the next chapter, you will learn how to programmatically do all the stuff you just did using the admin app. You will learn how to add, edit, and delete data from the database using query sets.

Questions

What are the two methods of creating an app in a Django project?

What is the base class of any model class?

How any model is registered in the admin app?

Which users can log in to the admin app of the project?

What are the access rights available to the superuser?

CHAPTER 5

Interacting with the Database Using Query Sets

Introduction

You have already manipulated the data of the present in the database using the admin panel. It is required to learn about query sets as you will be using these in the upcoming chapters. The query sets are written in the views. So, when the logic is implemented, based on the logic the corresponding query set is executed and the required DB operation is performed. To fetch the data from the database or to save data in the database, you must know how to interact with the database programmatically.

In this chapter, you will learn about how to manipulate the data programmatically. You will learn about query sets and their different functions. You will also learn how to fetch data from DB and how to insert new data into it.

Structure

Revisiting ORM

Query sets

Adding elements to your database

Manipulating elements of your database

Deleting elements of your database

Objectives

Understanding the concept of **Object Relational Mapping**

Understanding query sets.

Performing read-write operations using query sets.

ORM overview

Python provides a built-in interface for admin functionalities. It is more like an app. Now that you have a fully functional administration site to manage your blog's content, it's time to learn how to retrieve information from the database and interact with it. Django comes with a powerful database-abstraction API that allows you to create, retrieve, update, and delete objects easily. The Django ORM is compatible with MySQL, PostgreSQL, SQLite, and Oracle. Remember that you can define the database of your project by editing the DATABASES setting in the settings.py file of your project. Django can work with multiple databases at a time and you can even program database routers that handle the data in any way you like.

Query sets

In the previous chapter, you created a database and interacted with that database using the admin app. Now, you will interact with the same database programmatically using query sets. The QuerySet is basic commands which when evaluated can interact with the database. So, to perform database operations, you will have to write the Only writing the QuerySet will not do the work. The QuerySet can be written for different kinds of purposes but it will not interact with database unless it is evaluated.

You can perform different operations on a Performing these operations evaluates the QuerySet objects:

Iteration: You can write a QuerySet and iterate over it. For example, you have a database of books. You can write a QuerySet and iterate over the returned data to access each of the different records one at a time.

Slicing: This is similar to the slicing of arrays. You can slice a QuerySet just as you slice arrays or lists by using the index. Although slicing an unevaluated QuerySet will give an unevaluated Slicing an unevaluated QuerySet usually returns another unevaluated

`repr()`: This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API

interactively.

`len()`: This is used to get the length of the result list.

`list()`: This is used to save the results of the QuerySet in a list.

`bool()`: This is used to check whether the QuerySet has any result.

Some common methods used on query sets are as follows:

Filter

This function is used to provide conditions to the query sets. It will return the results matching the conditions provided.

Exclude

This is used to exclude the results not satisfying the conditions provided. It will return a set of results that do not qualify the mentioned conditions.

Latest

This will return the most recent object in the table out of the objects which satisfy the given conditions.

Update

This query is used to update the records in the table. It will return the no of rows that match to the condition which may or may not be equal to the number of rows updated depending on the value of data present already.

Delete

This is used to run a delete query on the table and it returns the number of objects deleted along with a dictionary with the number of deletions per object type. This query can be filtered while being executed. It should be applied to an already filtered query.

In Django, if there is a ForeignKey element, then the related element will also be deleted unless `on_delete = CASCADE` is changed in the model definition.

Get

This is used to get an object matching the given conditions. This function raises an exception `MultipleObjectsReturned` if more than one object is found.

These are some of the `QuerySet` operations you can perform over a database table. Mostly, you will use the `filter`, `update`, `get`, and `delete` operations.

[Adding elements to your database](#)

You have read enough about how to start writing query sets. Now, you will write query sets which will interact with your database created in the previous chapter. Perform the following steps:

Go to your project folder and check out the contents:

```
C:\Work\Project_root\mysite>dir
 Volume in drive C is Windows
 Volume Serial Number is 4883-E662

 Directory of C:\Work\Project_root\mysite

06-10-2019  23:23    <DIR>        .
06-10-2019  23:23    <DIR>        ..
06-10-2019  23:23            147,456 db.sqlite3
05-10-2019  13:26            647 manage.py
06-10-2019  22:37    <DIR>        mysite
06-10-2019  23:19    <DIR>        student
                2 File(s)        148,103 bytes
                4 Dir(s)   172,338,606,080 bytes free

C:\Work\Project_root\mysite>
```

Figure 5.1: Project directory

Now, go to your app folder and check out the contents:

```
C:\Work\Project_root\mysite>dir
 Volume in drive C is Windows
 Volume Serial Number is 4883-E662

 Directory of C:\Work\Project_root\mysite\student

06-10-2019  23:19    <DIR>          .
06-10-2019  23:19    <DIR>          ..
06-10-2019  22:42            135 admin.py
05-10-2019  18:26            94 apps.py
06-10-2019  23:19    <DIR>          migrations
06-10-2019  23:19            732 models.py
05-10-2019  18:26            63 tests.py
05-10-2019  18:26            66 views.py
05-10-2019  18:26            0 __init__.py
06-10-2019  23:19    <DIR>          __pycache__
                           6 File(s)   1,090 bytes
                           4 Dir(s)  172,321,579,008 bytes free

C:\Work\Project_root\mysite>
```

Figure 5.2: Opening app directory

You should have a similar folder structure. If yes, then we can move forward else you may want to re-execute the steps from the previous chapters as there might be an error.

Go back to your project folder and execute the python manage.py shell command:

```
C:\Work\Project_root\mysite>python manage.py shell
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.19
16 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more informa
tion.
(InteractiveConsole)
>>>
```

Figure 5.3: Opening Python console

Now, execute the following commands. See the following screenshot for reference.

The following command imports the StudentDetail model to your shell interpreter. The class properties for this model class will be available to you to use:

```
>>>from student.models import StudentDetail
```

The following command imports the datetime module so that we can provide the date of birth of students:

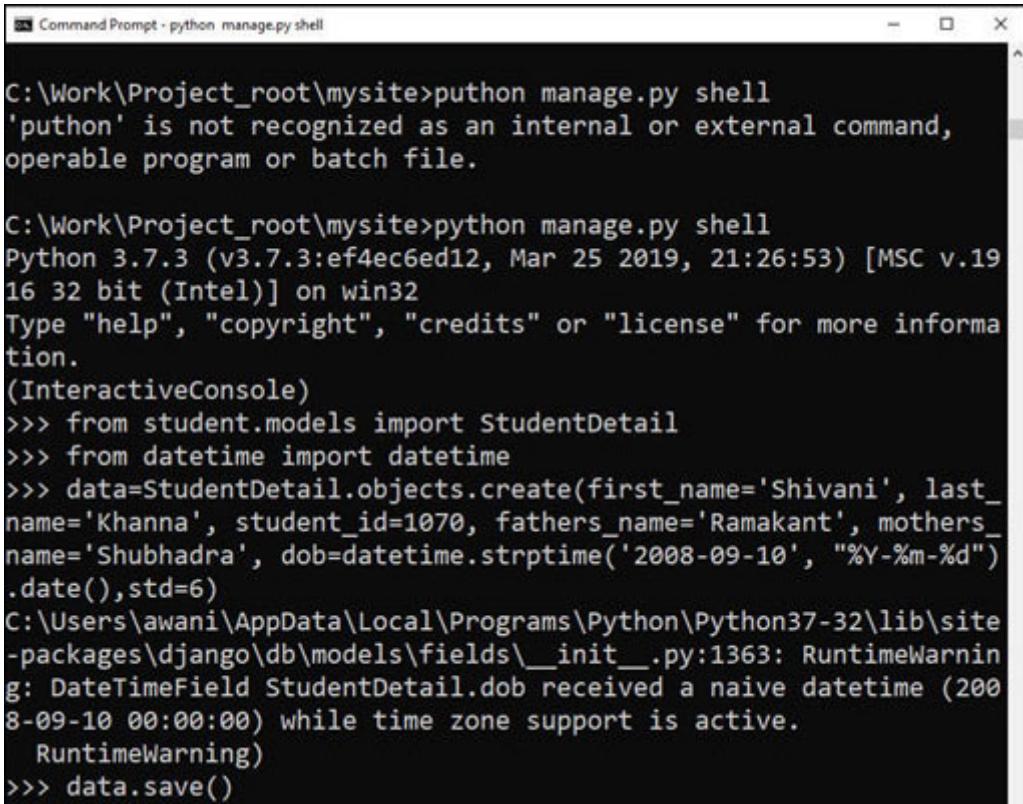
```
>>>from datetime import datetime
```

Now, you create an object of your model class StudentDetail. You need to provide the value for all the not NULL attributes of the StudentDetail model class as defined:

```
>>>data=StudentDetail.objects.create(first_name='Shivani',  
last_name='Khanna', student_id=1070, fathers_name='Ramakant',  
mothers_name='Shubhadra', dob=datetime.strptime('2008-09-10',  
"%Y-%m-%d").date(),std=6)
```

Finally, you save the object created in the database which is similar to making a commit in SQL:

```
>>>data.save()
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt - python manage.py shell". The command "python manage.py shell" is run, followed by a series of Python commands to create a new instance of a "StudentDetail" model. The command "data.save()" is executed, but a warning message is printed to the console: "RuntimeWarning: DateTimeField StudentDetail.dob received a naive datetime (2008-09-10 00:00:00) while time zone support is active." The Python version shown is 3.7.3.

```
C:\Work\Project_root\mysite>python manage.py shell
'python' is not recognized as an internal or external command,
operable program or batch file.

C:\Work\Project_root\mysite>python manage.py shell
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.19
16 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more informa
tion.
(InteractiveConsole)
>>> from student.models import StudentDetail
>>> from datetime import datetime
>>> data=StudentDetail.objects.create(first_name='Shivani', last_
name='Khanna', student_id=1070, fathers_name='Ramakant', mothers_
name='Shubhadra', dob=datetime.strptime('2008-09-10', "%Y-%m-%d")
.date(),std=6)
C:\Users\awani\AppData\Local\Programs\Python\Python37-32\lib\site
-packages\django\db\models\fields\__init__.py:1363: RuntimeWarnin
g: DateTimeField StudentDetail.dob received a naive datetime (20
08-09-10 00:00:00) while time zone support is active.
  RuntimeWarning)
>>> data.save()
```

Figure 5.4: Executing Python commands using query sets to add data to the database

You have inserted your data into the database. Now, you would want to see if the data is inserted into the database. Let's do that.

Execute the following commands and see the following screenshot for reference.

Create a list variable to store the result of the execution of a query set which fetches all the data in the table:

```
>>>all_students= Student.Detail.objects.all()
```

Now, all_students is a list that has a list of objects which were returned by the execution of the query set Let's check what all_students has:

```
>>>all_students
```

So, the last object entered is the object (7) at the 6th position in the list To see the data of that object, execute the following command:

```
>>>all_students[6].first_name
```

```
>>> all_students_list= StudentDetail.objects.all()
>>> all_students_list
<QuerySet [<StudentDetail: StudentDetail object (1)>, <StudentDetail: StudentDetail object (2)>, <StudentDetail: StudentDetail object (3)>, <StudentDetail: StudentDetail object (4)>, <StudentDetail: StudentDetail object (5)>, <StudentDetail: StudentDetail object (6)>, <StudentDetail: StudentDetail object (7)>]>
>>> all_students_list[6].first_name
'Shivani'
>>>
```

Figure 5.5: Checking data stored in the database using query sets

So, you see the data of Shivani is inserted as you wanted.

Points to note here are:

You need to provide data for all the attributes which are not nullable in the model class definition. The value of attributes should be according to the definition in

When you create an object of the model class, it creates a row in the table. To edit the attributes/columns of the table, you need to access it through the particular object.

After saving the object, you can also go to the admin page and see the new object created there along with the objects created in the previous chapter.

You have read how Django uses ORM for interacting with databases. How to write database queries using the concepts of ORM. Now, you will read about how to instruct ORM to perform the update or manipulate the existing data in the database.

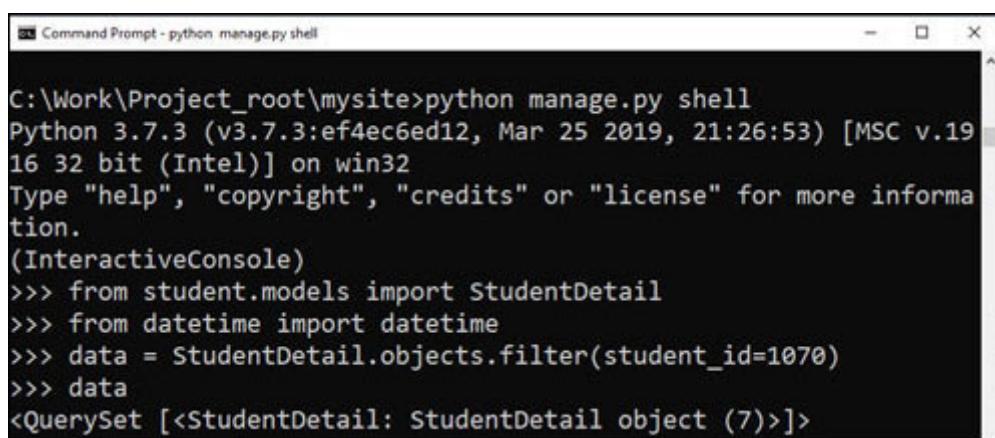
Manipulating elements of your database

You learned how to create new rows in DB and fetch data from DB using query sets. Now, you will fetch a particular data, modify it, and save it back to DB. Execute the following commands and take a look at the following screenshot:

```
>>> from student.models import StudentDetail
```

Pick any condition based on which you want to filter. Here, let's say we did not enter the full name of Shivani's parents in the previous insertion and so we need to make an update to Shivani's records. So, we fetch Shivani's DB record from the table in an object using the following query set.

```
>>> data = StudentDetail.objects.filter(student_id=1070)
>>> data
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt - python manage.py shell". The window displays the following Python code and its output:

```
C:\Work\Project_root\mysite>python manage.py shell
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from student.models import StudentDetail
>>> from datetime import datetime
>>> data = StudentDetail.objects.filter(student_id=1070)
>>> data
<QuerySet [<StudentDetail: StudentDetail object (7)>]>
```

Figure 5.6: Executing filter command

So, data is a list of objects with only one object as the given condition in the query set student_id=1070 matches with only one record. Now, check what are Shivani's parent's names in the database:

```
>>>data[0].mothers_name  
>>>data[0].fathers_name
```

```
>>> data[0].mothers_name  
'Shubhadra'  
>>> data[0].fathers_name  
'Ramakant'
```

Figure 5.7: Fetching data from objects using query sets

We need to modify their names and add the surname. For this, we need to write an update query set or as mentioned in the preceding section, as iterating over a query set evaluates it. Let's try both. Execute the following command. If it returns 1, your update is successful.

```
>>>StudentDetail.objects.filter(student_id=1070).update(mothers_name='Shubhadra Khanna')
```

Check the preceding update by executing the following command. It should have the surname:

```
>>>data[0].mothers_name
```

```
>>> StudentDetail.objects.filter(student_id=1070).update(mothers_
name='Shubhadra Khanna')
1
>>> data[0].mothers_name
'Shubhadra Khanna'
```

Figure 5.8: Fetching data from objects using query sets

Now, let's try iterating over the query set. We have data which is a list containing the required object. You can iterate over this list and access the required attribute and update. Iterating will evaluate the query set and changes will be updated:

```
>>>for obj in data:
obj.fathers_name="Ramakant Khanna"
obj.save()
```

Check the preceding update by executing the following command. It should have the surname:

```
>>>data[0].fathers_name
```

```
>>> for obj in data:
...     obj.fathers_name='Ramakant Khanna'
...     obj.save()
...
>>> data[0].fathers_name
'Ramakant Khanna'
>>>
```

Figure 5.9: Looping through objects using query sets

Note that once updated, you can go to the admin page and see changes reflected. As discussed earlier, you can use `objects.get(condition)` instead of `objects.filter(condition)`. Using `get` will return the object directly and using a filter will return a list of objects matching the condition.

Deleting elements of your database

Deleting a record is similar to modifying a record. You need to fetch the records you want to delete. If it's a single record, you can use get() else Just fetch the record and use the delete function. Perform the given steps as shown in the following screenshot:

```
>>> all_students_list= StudentDetail.objects.all()  
>>> all_students_list
```

```
>>> all_students_list= StudentDetail.objects.all()  
>>> all_students_list  
<QuerySet [<StudentDetail: StudentDetail object (1)>, <StudentDetail: StudentDetail object (2)>, <StudentDetail: StudentDetail object (3)>, <StudentDetail: StudentDetail object (4)>, <StudentDetail: StudentDetail object (5)>, <StudentDetail: StudentDetail object (6)>, <StudentDetail: StudentDetail object (7)>]>  
>>>
```

Figure 5.10: Fetching all objects

Let's check the details of the 4th student:

```
>>>all_students_list[3].first_name  
>>>all_students_list[3].student_id
```

```
>>> all_students_list[3].first_name  
'Suresh'  
>>> all_students_list[3].student_id  
'1040'  
>>>
```

Figure 5.11: Fetching data using query sets

Let's delete Suresh's records:

```
>>> data = StudentDetail.objects.get(student_id=1040)  
>>> data.delete()
```

```
>>> data = StudentDetail.objects.get(student_id=1040)  
>>> data.delete()  
(1, {'student.StudentDetail': 1})  
>>>
```

Figure 5.12: Deleting data using query sets

Now, check the list of all students. You will not find Suresh's records:

```
>>> all_students_list= StudentDetail.objects.all()  
>>> all_students_list
```

```
>>> all_students_list= StudentDetail.objects.all()  
>>> all_students_list  
<QuerySet [<StudentDetail: StudentDetail object (1)>, <StudentDetail: StudentDetail object (2)>, <StudentDetail: StudentDetail object (3)>, <StudentDetail: StudentDetail object (5)>, <StudentDetail: StudentDetail object (6)>, <StudentDetail: StudentDetail object (7)>]>  
>>>
```

Figure 5.13: Fetching data from the database

You will find no records of the 4th object as the object is deleted.

Conclusion

In this chapter, you learned about programmatically interacting with the database using Python code on the Python console. These commands will be used in your programs further to communicate with the database. You will be writing these query sets in views.py file to create logic and perform CRUD operations on user demand and requirement. You learnt an important concept in this chapter. Now, you will be able to interact with the database dynamically based on the conditions you receive in the request and you will be able to make database transitions as needed.

In the next chapter, you will be working more on views.py, and models.py and enhance your project where it makes more sense. Then, later you will learn about these in more detail.

Questions

What is ORM?

How do you apply conditions on which record is to be fetched from DB?

Which query set is used when you need to fetch any object which matches a given condition?

What is the use of the exclude query set?

What is the use of the repr query set?

CHAPTER 6

Enhancing Your Project

Introduction

Before you aim for a bigger project, you must see the bigger picture of a small project. This will help you in making analogies and keeping the components segregated and you would feel in control. You will see how to set up url-patterns in the urls.py file. You will see what kinds of classes are present in the models.py file. You will read about the view functions and see how these functions render templates. Consider this as a mini project.

This chapter will give you a thorough brief of the different components of Django. You will see the architecture of working. Once you have knowledge of how a project works and what type of code is written in the Django project files, the upcoming chapter will make more sense.

Structure

Understanding requirements

Making models

Making views

Setting up urls.py

Making templates

Conclusion

Objectives

Understanding the requirements of the project.

Understanding the role of the urls.py file.

Understanding how to convert requirements into model class.

Understanding how views connect with urls.py, models.py and templates.

Understanding requirements

In your project, you already have a model. Let's add some more models and other functionalities. We will create more models, write view functions to perform certain operations, and set up a basic frontend for your application. What we need is a simple application that gives a page where people can see the list of students enrolled in the school. This is the page where the user can add students and a page where the user can delete or modify students' details.

Before going ahead, let's take a look at what you are going to make in this mini project.

Screen 1: A page where the list of all the students is shown in the following screenshot:

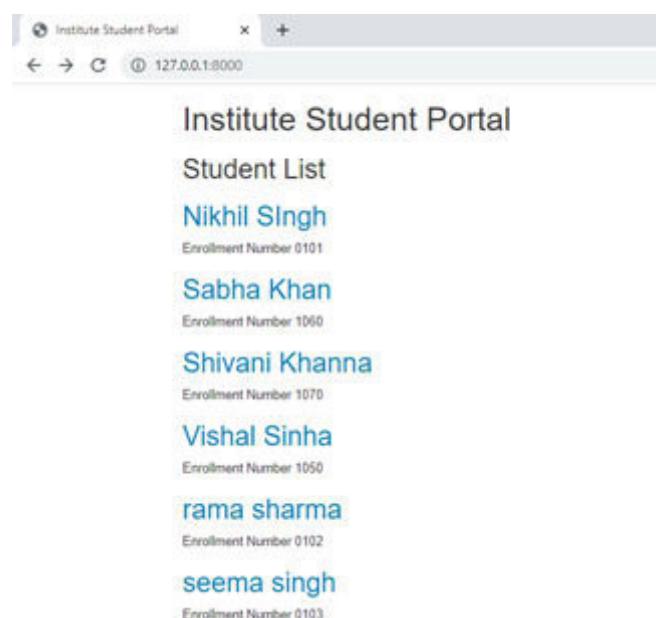


Figure 6.1: Output screen of the mini-project

Screen 2: A page where all the details of any specific student can be seen in the following screenshot:

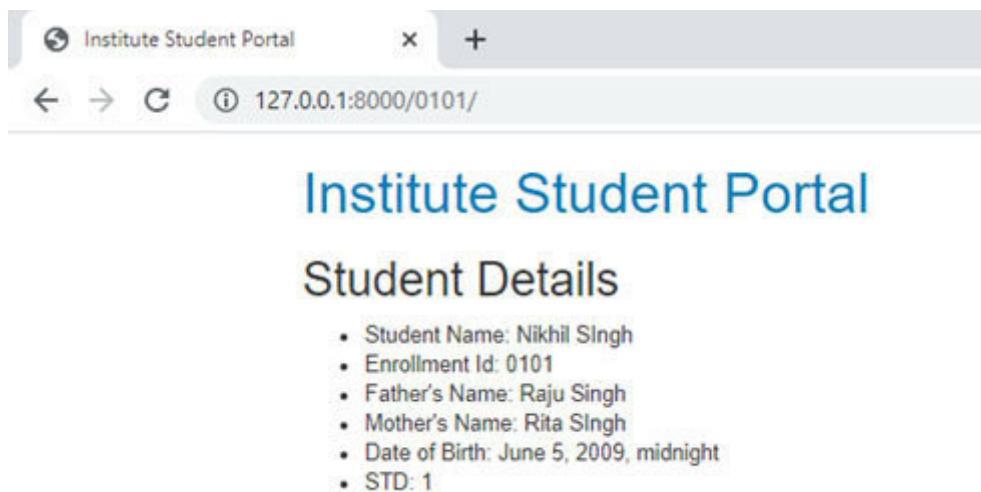
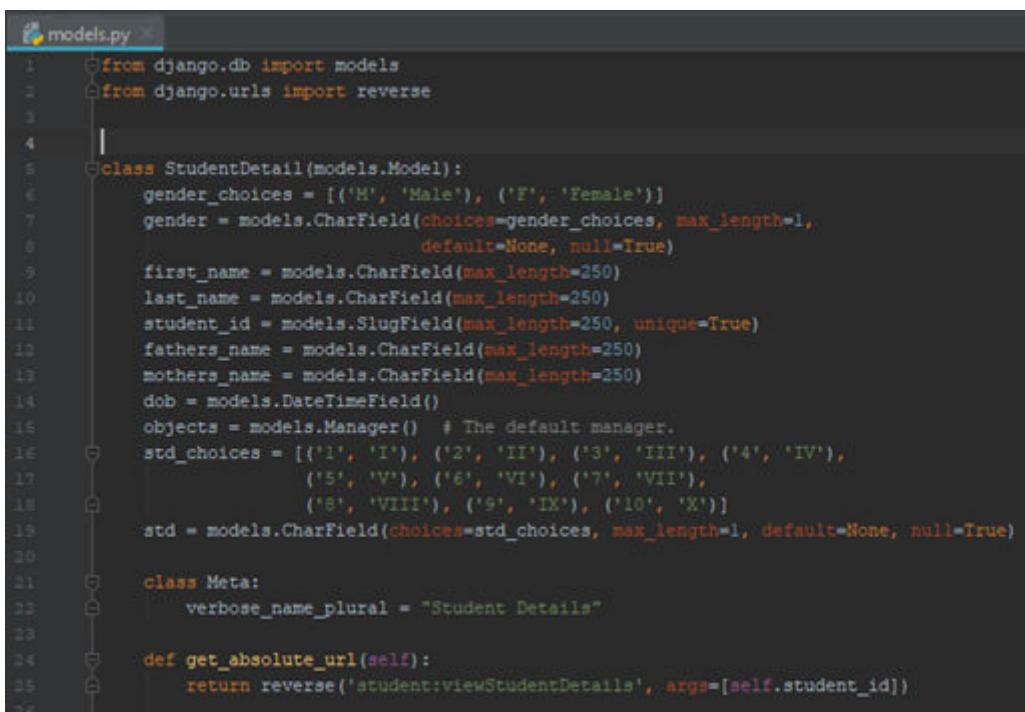


Figure 6.2: Output screen of the mini-project

We will create these two screens for now. The focus should be on understanding the role and location of different components.

Making Models

We already have a model that can have the data of students enrolled in the school. It contains personal information of students like name, parents' name, date of birth, etc. For now, let's work on this data. We already added some data to our database. The following screenshot displays a model that we created earlier:



```
models.py
1  from django.db import models
2  from django.urls import reverse
3
4  class StudentDetail(models.Model):
5      gender_choices = [('M', 'Male'), ('F', 'Female')]
6      gender = models.CharField(choices=gender_choices, max_length=1,
7                                  default=None, null=True)
8      first_name = models.CharField(max_length=250)
9      last_name = models.CharField(max_length=250)
10     student_id = models.SlugField(max_length=250, unique=True)
11     fathers_name = models.CharField(max_length=250)
12     mothers_name = models.CharField(max_length=250)
13     dob = models.DateTimeField()
14     objects = models.Manager() # The default manager.
15     std_choices = [(1, 'I'), (2, 'II'), (3, 'III'), (4, 'IV'),
16                     (5, 'V'), (6, 'VI'), (7, 'VII'),
17                     (8, 'VIII'), (9, 'IX'), (10, 'X')]
18     std = models.CharField(choices=std_choices, max_length=1, default=None, null=True)
19
20     class Meta:
21         verbose_name_plural = "Student Details"
22
23     def get_absolute_url(self):
24         return reverse('student:viewStudentDetails', args=[self.student_id])
```

Figure 6.3: Models.py file

We will be using the preceding model. Make sure this model is present in your project. There are two approaches to develop any Django project. The first is to start with Models (database layer)

followed by creating views. Once the views are ready, then you can write your urls.py mapping to the views. Finally, templates should be designed as per the urlspattens in urls.py and views in This approach can be referred to as backend focused. This approach is to be chosen when the requirements of the database are clear. We will be using this approach.

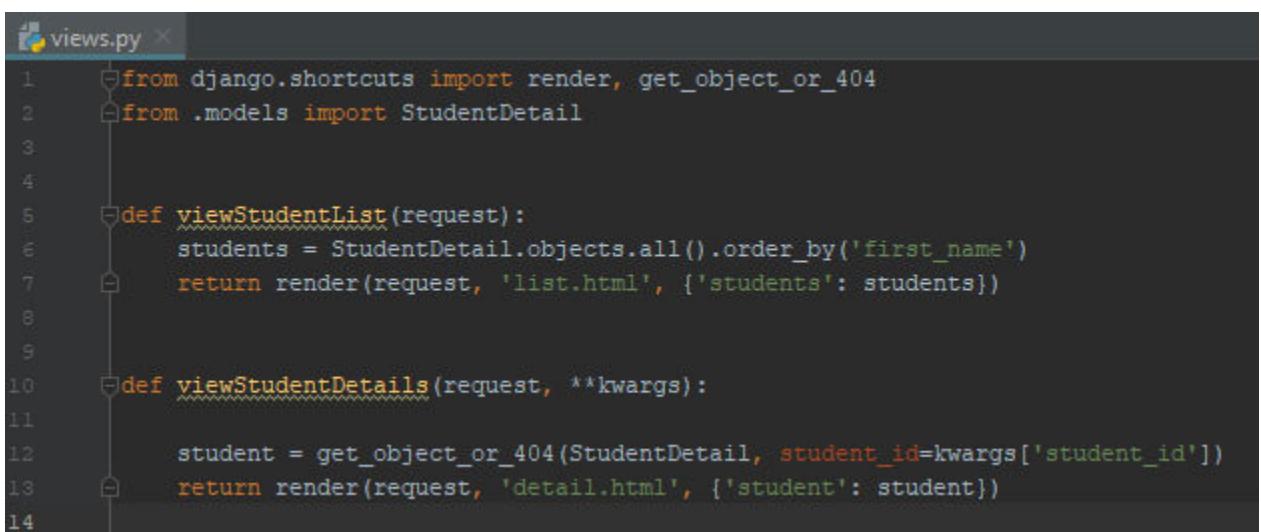
The other approach is to design the templates first, followed by urls.py and then, views and models. This approach can be chosen when the clarity of the frontend is better than the backend.

Creating views

Views are responsible for performing the actions. Any request made from the frontend is resolved at urls.py and then the corresponding view is executed. Hence, for every expected request, we need to create a view. As of now, our application is supposed to add new students, update student details, delete data, and fetch student details from the database and display them on the page. We will need a view for each of these operations.

There are two types of views: function-based views and class-based views. Class-based views are more advanced and easy to use but only when understood properly. You will learn that in the upcoming chapters. For now, we will use function-based views.

Write the following code in your views.py file:



```
views.py ×
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3
4
5  def viewStudentList(request):
6      students = StudentDetail.objects.all().order_by('first_name')
7      return render(request, 'list.html', {'students': students})
8
9
10 def viewStudentDetails(request, **kwargs):
11
12     student = get_object_or_404(StudentDetail, student_id=kwargs['student_id'])
13     return render(request, 'detail.html', {'student': student})
14
```

Figure 6.4: views.py file

Here, we have created two views. One view function to see the list of all the students in the institute and the other view to see the details of a student Now, let's take a look at the preceding code.

You have a requirement to display a list of all the students in the institute. To get this list, you need to fetch all the students in the StudentDetail table. Recall the query set for such a query from the previous chapter: So, the students variable in the viewStudentList function contains the list of all the objects of the StudentDetail class or simply you can say, it contains the details of all the students in the StudentDetail table.

Once we have the data which is to be displayed, the next step is to return it with a suitable template. This is done using the render function. It takes the request for which the view sends the response, the template, and a dictionary.

This dictionary contains key : value Key is the variable in the template and the value is the data fetched from the database. So, the data fetched from the database is sent to the template and the template is rendered in response to the request received. Here, the task of the viewStudentList view function is to send a dictionary to the template This dictionary should contain a list of all the objects of the StudentDetail class. Now, it's the job of the template to display it on the webpage.

When the render function is executed, the template is loaded. The template uses the data it receives from the dictionary. The template has the Python code to generate dynamic HTML needed to display the data in the required format. Now, the other expectation is to see the details of one student. Thus, when the link for that student is clicked, a specific url-pattern is created on the browser. A request is created with those patterns. The pattern matches the second url-pattern defined in the student/urls.py file. Thus, the view function is invoked.

The request is received from the urls.py to views function contains extra arguments. These arguments are used to identify the object which needs to be fetched (the student for whom details are to be fetched). Once the details of that object are obtained, it is passed to the template and the template is rendered in response. Again, it's the job of the template to display the data in a proper format.

Configuring urls.py

The URL formed on the browser when any link on a Django application is clicked decides the flow of the request. The URL is matched to a list of url-patterns in the urls.py file of the Django project. Each of these patterns has a corresponding view function. That view function is invoked and a response is generated.

Now, as you know, one project can have multiple apps, so to keep the things segregated we need to create a separate urls.py file for every app inside the app folder. Then, include the url-patterns written in the app's urls.py file into the project's urls.py file. The app's urls.py file is shown in the following screenshot. It contains a list of URLs patterns. Each element of the list is a function call containing three arguments: URL pattern, view function, and namespace. The URL pattern is matched with what is formed on the browser and the corresponding view is invoked.

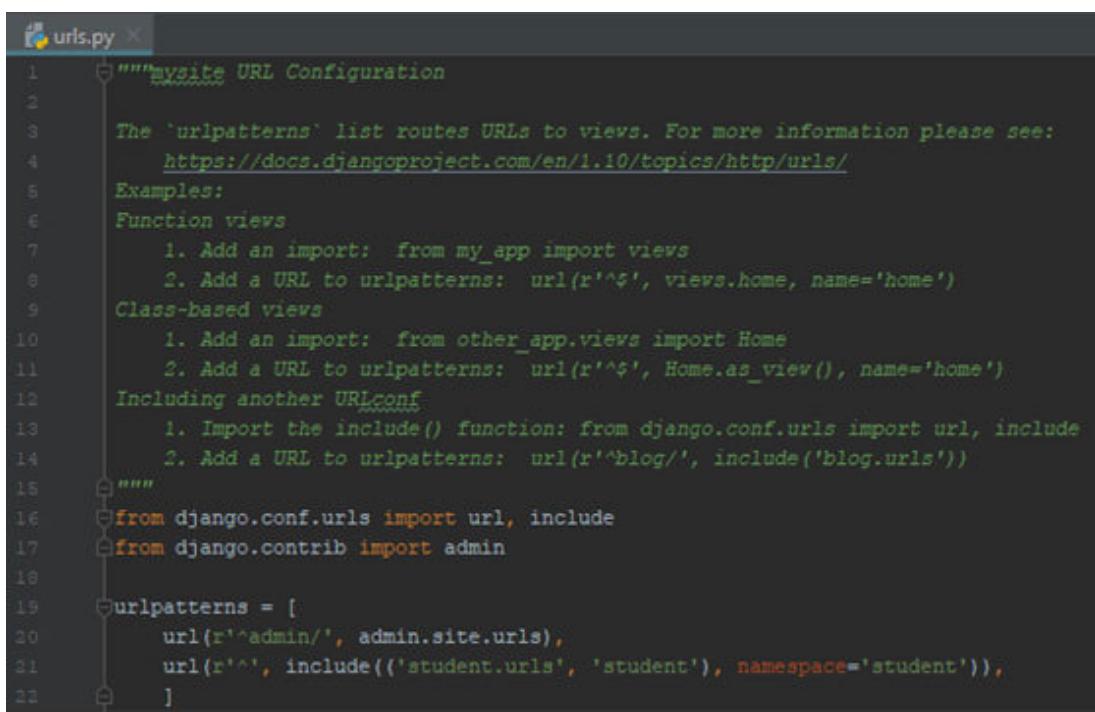


```
urls.py
1  from django.conf.urls import url
2  from . import views
3
4
5  urlpatterns = [
6      url(r'^$', views.viewStudentList, name='viewStudentList'),
7      url(r'^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
8  ]
```

url pattern *view function*

Figure 6.5: App's urls.py file

The following screenshot displays the urls.py file of the project. Just like the preceding file, it also contains a list of URL patterns. The URL in the request received from the browser is passed in this file. The URL is matched with the patterns present in this file. You can see that we have included the urls.py file of the app in the urls.py file of the project. So the URL patterns of the app's urls.py file will also be matched here itself. If any of the patterns are matched with the request from the browser, then the corresponding view of the app will be invoked:



The screenshot shows a code editor window with the file 'urls.py' open. The code is as follows:

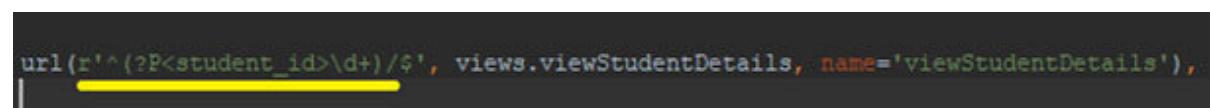
```
1  """mysite URL Configuration
2
3  The 'urlpatterns' list routes URLs to views. For more information please see:
4      https://docs.djangoproject.com/en/1.10/topics/http/urls/
5  Examples:
6      Function views
7          1. Add an import: from my_app import views
8              2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')
9      Class-based views
10         1. Add an import: from other_app.views import Home
11             2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')
12     Including another URLconf
13         1. Import the include() function: from django.conf.urls import url, include
14             2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))
15
16 """
17 from django.conf.urls import url, include
18 from django.contrib import admin
19
20 urlpatterns = [
21     url(r'^admin/', admin.site.urls),
22     url(r'^', include(('student.urls', 'student'), namespace='student')),
23 ]
```

Figure 6.6: Project's urls.py file

Note that the URL matching is done sequentially. So, if the URL matches with the first url-pattern in the urlpatterns list, it will not be matched further and the view of the first url-pattern will be invoked. If the pattern on the browser does not match with the first pattern, it will try to match it with the second and so on. Here, in the place of the second pattern, we have included the url-patterns of our app; thus, it will start matching with the url-patterns present in

the app's urls.py file. If there is a match, the corresponding view will be invoked. If there is no match in the app's urls.py file, the interpreter will be back to the project's urls.py file and look for the third url-patter to match.

You must be confused about how the URLs on the browser are generated and what to write as expected patterns. The patterns you see in the following screenshot are regular expressions. It's a module of Python known as *regex* or *re* module:



```
url(r'^^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
```

A screenshot of a code editor showing a single line of Python code. The line contains a URL pattern definition using the 'url' function from Django's 'urls.py' module. The pattern is 'r'^^(?P<student_id>\d+)/\$'. The word 'views' and 'viewStudentDetails' are imports from another module. The 'name' parameter is also part of the URL pattern. The entire line is highlighted with a yellow background.

Figure 6.7: a sample of url-pattern in urlpatterns list

In case none of the url-patterns matches, the program will display an error page not found. For example:

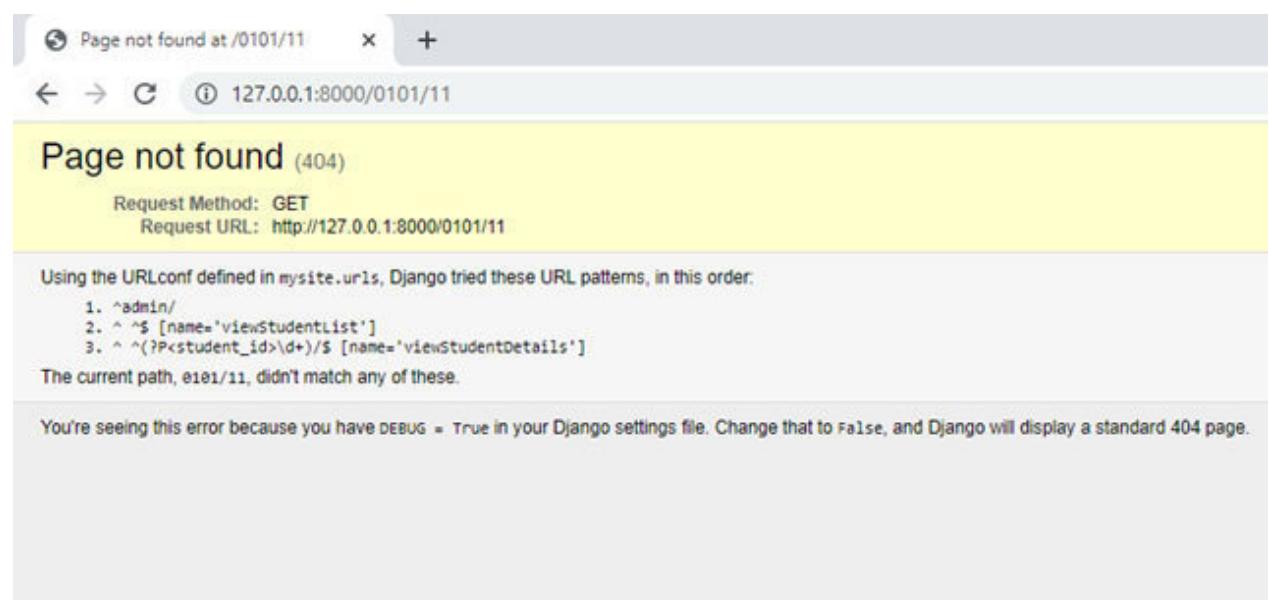


Figure 6.8: Error page

This is a 404 page. It may look different in different versions of Django. You will read more about URLs and regex in the upcoming chapter. For now, write the code accordingly in your project.

Making templates

Once the URL is matched and the corresponding view function is invoked, it returns a template in return. This template is rendered in the browser as a result. By now, you must have come to know that it is the job of this template to display the data it receives in a proper format. Now, we are about to do some frontend coding. So, go to the project directory in file explorer. Open your app folder. Create folders named static and Now, open the static folder and create two folders in it named js and

Then, inside the js folder create a file student.js and in the student folder, create Inside the the folder creates three HTML files names base.html, and

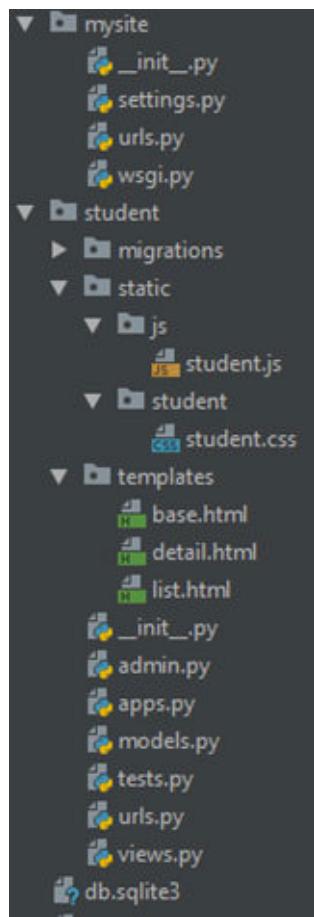


Figure 6.9: Project's files and folder

Get back to PyCharm and your project structure will look like the following screenshot. Now, open the settings.py file of your project and edit the following values as shown in the following screenshot:

```
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATE_DIR = os.path.join(BASE_DIR, 'student/templates')
```

Figure 6.10: Adding path to templates in the settings.py file

Update the TEMPLATES dictionary and make it look like the following screenshot:

```
ROOT_URLCONF = 'mysite.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR, ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Figure 6.11: Configuring Template settings.

Update the STATIC_URL and STATIC_ROOT variables as shown in the following screenshot:

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

Figure 6.12: Setting a static location and URL

Once you have completed editing your settings.py file, follow the given sets. Keeping the scope of the book in mind, we will not discuss the CSS and JS code. The student.js file will be blank for now as we are not running any JavaScript as of now.

Open the student.css file and type the following code it. If you have any experience in CSS, then it's great. You can write your CSS, the site may look different though. If you do not have any CSS knowledge, it's nothing to worry about. Just follow along:

```
# student.css
1 .techfont{
2     font-family: 'Russo One', sans-serif;
3     font-size: 1.5em;
4     margin-bottom: 10px;
5 }
6 .postdate{
7     text-align: center;
8 }
9 .posttitle{
10    font-family: 'Russo One', sans-serif;
11    font-size: 3em;
12    text-align: center;
13 }
14 .postcontent{
15    font-family: 'Montserrat';
16    font-size: 1.5em;
17 }
18 .centerstage{
19     margin-left: auto;
20     margin-right: auto;
21 }
22 .btn-comment{
23     position: absolute;
24     right: 0px;
25 }
26 .bigbrand{
27     font-size: 1.5em;
28 }
29 /*COLOR CHANGER*/
30 /*Credit and Source: http://codepen.io/the
```

```
# student.css
31 /* Credit and Source: http://codepen.io/the
32 .loader{
33     filter:hue-rotate(0deg);
34     color: linear-gradient(45deg,#0f8,#08f);
35     animation:hue 5000ms infinite linear;
36 }
37 @keyframes spinify {
38     0% {
39         transform: translate(0px,0px);
40     }
41     33% {
42         transform: translate(0px,24px);
43         border-radius:100%;
44         width:10px;
45         height:10px;
46     }
47     66% {
48         transform:translate(0px,-16px);
49     }
50     88% {
51         transform:translate(0px,4px);
52     }
53     100% {
54         transform:translate(0px,0px);
55     }
56 }
57 @keyframes hue{
58     0%{filter: hue-rotate(0deg);}
59     100%{filter: hue-rotate(360deg);}
60 }
```

Figure 6.13: Code of student.css file.

Now, open the base.html file in templates and make it look like the following screenshot. This is the base template that will be inherited by other templates. The block content tag contains the content of the page:

```
base.html
1  <!DOCTYPE html>
2  (% load staticfiles %)
3  <html>
4      <head>
5          <title>Institute Student Portal</title>
6          <!-- Latest compiled and minified CSS -->
7          <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
8              integrity="sha384-BVYiiSIFeKIdGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
9              crossorigin="anonymous">
10     <body class='loader'>
11         <div class="content container">
12             <div class="row">
13                 <div class="col-md-8">
14                     <div class="blog_posts">
15                         (% block content %)
16                         (% endblock %)
17                     </div>
18                 </div>
19             </div>
20         </div>
21         (# SCRIPTS#)
22         <script type="text/javascript" src="(% static 'js/blog.js' %)"></script>
23     </body>
24 </html>
```

*This is scope for other
templates which inherit this
base template*

Figure 6.14: *base.html file*

The following is the HTML code of *base.html* for reference:

```
html>
{%
    load staticfiles
    rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-BVYiiSIFeKIdGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous"
    class='loader'
}
<div class="content container">
    <div class="row">
        <div class="col-md-8">
            <div class="studentdetails">
```

```
{% block content %}  
{% endblock %}  
{# SCRIPTS#}
```

Now, open list.html and write the following code:

```
{% extends "base.html" %}  
{% block title %}Our Institute{% endblock %}  
{% block content %}
```

>Institute Student Portal

Student List

{% for student in students %}

```

    href="{{student.get_absolute_url}}">>
{{student.first_name}} {{student.last_name}}
```

class="date">> Enrollment Number {{student.student_id}}

{% endfor %}

{% endblock %}

The list.html file should look like the following screenshot:

The screenshot shows the list.html file in a code editor. The code is a Django template with the following structure:

```

1  {% extends "base.html" %}
2  {% block title %}Our Institute{% endblock %}
3  {% block content %}
4      <h1>Institute Student Portal</h1>
5      <h2>Student List</h2>
6      {% for student in students %}
7          <h2>
8              <a href="{{ student.get_absolute_url }}">
9                  {{ student.first_name }} {{ student.last_name }}
10             </a>   This is the link to the specific student
11         </h2>
12         <p class="date"> Enrollment Number {{ student.student_id }} </p>
13     {% endfor %}
14     {% endblock %}|
```

Annotations explain the code:

- The method is defined in models.py file. This will be the url generated at the browser**: Points to the {{ student.get_absolute_url }} line.
- The list received from viewsFunction via dictionary**: Points to the {{ student.first_name }} {{ student.last_name }} line.
- This is the link to the specific student**: Points to the line.

Figure 6.15: list.html file

Here, you see a for loop iterating through the list sent to the template by the view function. So, you have a Python code running on an HTML file which is creating URLs dynamically by invoking the method written in the model class. Amazing isn't it!!

Now, open the detail.html file and write the following code in it:

```
{% extends "base.html" %}  
{% block title %} Student Details {% endblock %}  
{% block content %}
```

[Institute Student Portal](#)

Student Details

- Student Name: {{student.first_name}} {{student.last_name}}
 - Enrollment Id: {{student.student_id}}
 - Father's Name: {{student.fathers_name}}
 - Mother's Name: {{student.mothers_name}}
 - Date of Birth: {{student.dob}}
 - STD: {{student.std}}
- {% endblock %}

This is how the detail.html file should look like:

```
1  {% extends "base.html" %}  
2  {% block title %} Student Details {% endblock %}  
3  {% block content %}  
4  <h1>  
5      <a href="/">Institute Student Portal</a>  
6  </h1>  
7  <h2>Student Details</h2>  
8  <ul>  
9      <li>Student Name: {{ student.first_name }} {{ student.last_name }}</li>  
10     <li>Enrollment Id: {{ student.student_id }}</li>  
11     <li>Father's Name: {{ student.fathers_name }}</li>  
12     <li>Mother's Name: {{ student.mothers_name }}</li>  
13     <li>Date of Birth: {{ student.dob }}</li>  
14     <li>STD: {{ student.std }}</li>  
15  </ul>  
16  {% endblock %}|
```

*Details received from
viewStudentDetails
function via the dictionary*

Figure 6.16: detail.html file

Here, you have all the details of the student as an object fetched from the database. Now, the project is ready. Let us test it.

Testing

Now, go to the terminal, activate your environment, and execute the following commands:

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

Now, open your browser and hit the server showing in your terminal Here are the output screens, that is, the home page:

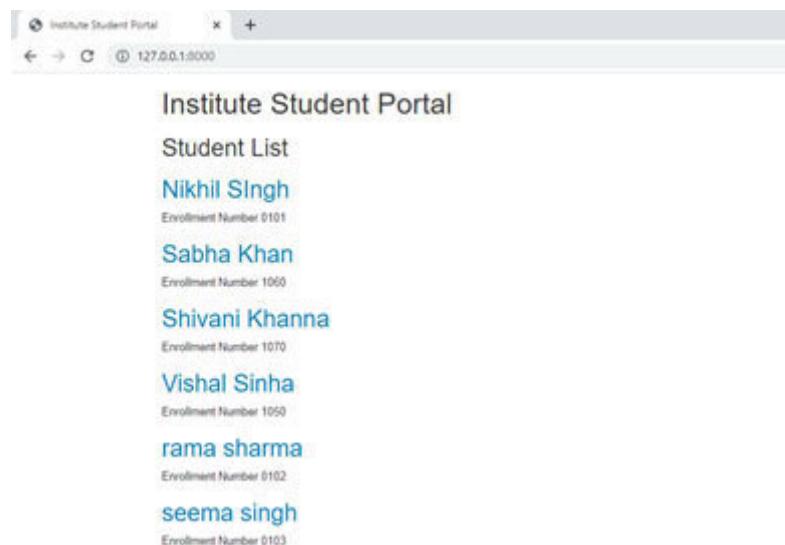


Figure 6.17: The homepage of your mini project

Here, you can see the list of students being displayed. The template here is list.html rendered by the view StudentList.html

which was invoked by the first url-pattern as it matched with the URL on the browser.

Now, click on any student and the student detail page will open:

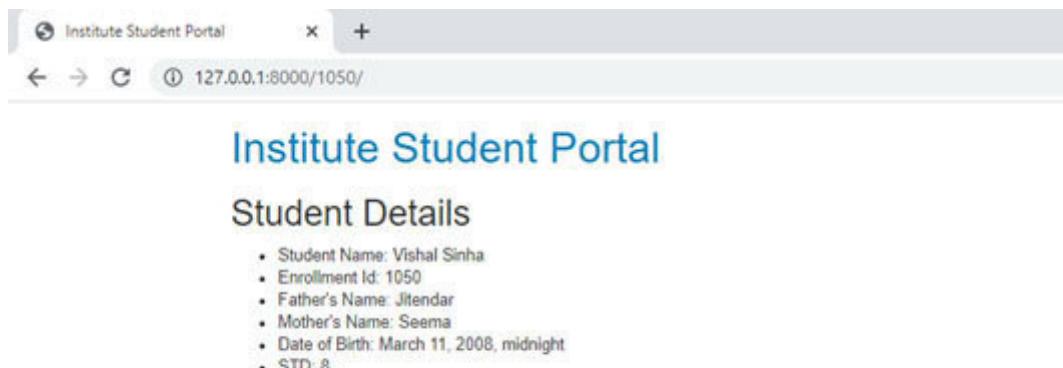


Figure 6.18: Details page

Try this out. If you face any issue, make sure you are using the correct combination of versions of packages installed.

Conclusion

In this chapter, you started with gathering requirements. Based on that, you worked on model classes and prepared a database for your project. You coded the logic into the views.py file and made changes to the setting.py file to meet the requirements. You prepared templates to create HTML pages to display on the webpage. You learnt how to create a Django project. You also learnt about the code in different components of a Django project. Now, you have a working project in Django that can display a list of students' details of students.

In the upcoming chapters, you will read about each of these components in detail and then you will work on a larger project. In the next chapter, you will focus on models and different features and options present in it. You will read about fields, field types, relationships, etc. You will also learn how to save your project and keep a copy of it as you may need it for future references.

Questions

What does a model class signify in a database project?

What is the main role of the views.py file?

How is a view file connected to templates?

Which component of Django is responsible for connecting the request from the user to the corresponding view?

What are

How does the template get the data from the database to display on the page?

CHAPTER 7

Understanding Models

Introduction

Now, we will start digging into different components of Django in detail before we aim for a comparatively larger project. It is important that you have sufficient theoretical knowledge of these components before you try out their features.

In this chapter, you will learn more about the models of Django. You created a model class and used it in your project to store and retrieve data. Django provides a lot of features. The official Django documentation on models is over five hundred pages. Here, you will read about some of them which are used frequently and important to start developing an application in Django.

It is necessary to have an understanding of how model classes and their fields and field options are used to generate tables in the database by migrating models with relationships between them like one-to-one, many-to-many, and many-to-one.

Structure

Introduction

Model fields

Field options

Meta options

Model methods

Relationships

Connecting models of different apps

Objectives

Understand what model fields are.

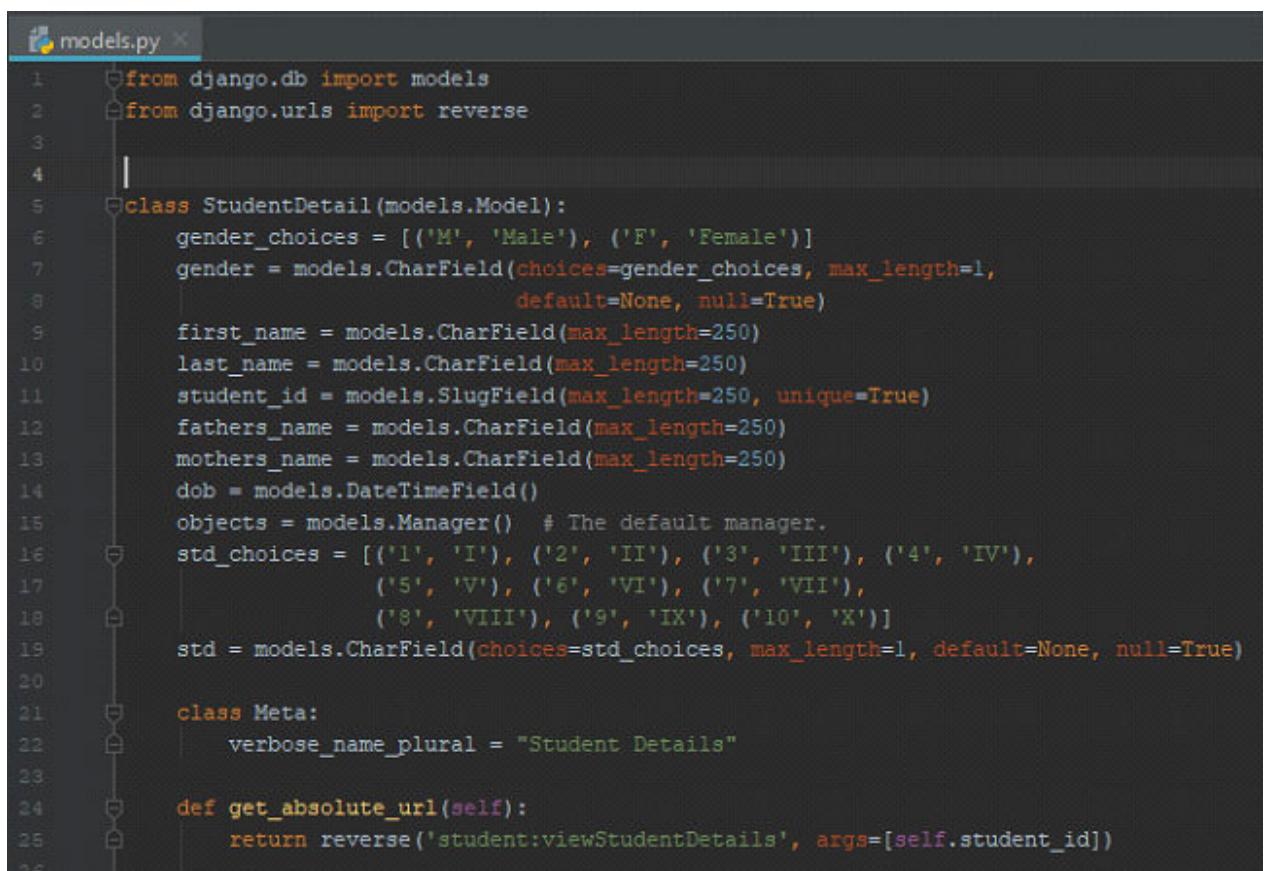
Understanding the use of field options available in different fields.

Understanding the use of metaclasses in models.

Understanding how the relationship between models is implemented.

Introduction to models

In the previous chapter, you worked on a project which had the models.py file. You used a model class to create a table in the database. Further, you manipulated the data in that table by using the admin app and query sets. Let's pull out that model.py file from your project from the previous chapter and see what has been used already:



```
models.py ×
1  from django.db import models
2  from django.urls import reverse
3
4
5  class StudentDetail(models.Model):
6      gender_choices = [('M', 'Male'), ('F', 'Female')]
7      gender = models.CharField(choices=gender_choices, max_length=1,
8                                  default=None, null=True)
9      first_name = models.CharField(max_length=250)
10     last_name = models.CharField(max_length=250)
11     student_id = models.SlugField(max_length=250, unique=True)
12     fathers_name = models.CharField(max_length=250)
13     mothers_name = models.CharField(max_length=250)
14     dob = models.DateTimeField()
15     objects = models.Manager() # The default manager.
16     std_choices = [(1, 'I'), (2, 'II'), (3, 'III'), (4, 'IV'),
17                    (5, 'V'), (6, 'VI'), (7, 'VII'),
18                    (8, 'VIII'), (9, 'IX'), (10, 'X')]
19     std = models.CharField(choices=std_choices, max_length=1, default=None, null=True)
20
21     class Meta:
22         verbose_name_plural = "Student Details"
23
24     def get_absolute_url(self):
25         return reverse('student:viewStudentDetails', args=[self.student_id])
```

Figure 7.1: The models.py file from the project

At a glance, one can say that the models.py file may contain import statements and the model class es. Further, the model class contains several fields, subclasses, and methods. This is a very basic structure of a models.py file. It can contain several model classes. Further, those classes can have several methods and subclasses. Let's look at them in more detail.

Model fields

As discussed earlier, model classes are like database tables and model fields are like table columns. It is a mandatory part of a model. In the preceding model, the first_name, last_name, etc. are model fields. There are several types of model fields like integer, varchar, text, etc. just like the type of columns/attributes of a database table. Here are some of the field types:

AutoField: This is an auto-generated field. It's like an inbuilt primary key of the table. If any of the fields in a model class is not set as a primary key, then Django creates this field. The value stored in this field is an incremental integer that increments with every new object of the model class created (row added to the table). You can use this field in your views. The name of this field by default is `id`. In your model, you haven't defined any primary key, thus you have an auto-generated primary key i.e. `id` and you can use it as follows:

```
student = StudentDetail.objects.all().filter(id=12): This will return the student object whose id(primary Key) is 12. If you wish to give this auto-generated primary key a different name, you can do that by adding the following field in your model class:
```

```
my_primary_key = models.AutoField(primary_key=True)
```

BooleanField: It is a True/False field. Example:

```
statement = models.BooleanField()
```

CharField: You have used this field in your model. It is like the char attribute of a database table. It can store text. For large texts, you can use You must be aware of the limitations of the database you configured in your settings.py file and should keep the max_length of this field accordingly.

DateField: This stores the date as an instance of the datetime.date class. Example:

```
date = DateField(auto_now=False, auto_now_add=False, **options)
```

If auto_now is true, the date is updated whenever the object is updated. If auto_now_add is true, the date is set to the date when the object is created.

DateTimeField: This stores the date-time stamp as an instance of the datetime.datetime class. Example:

```
date = DateTimeField (auto_now=False, auto_now_add=False,  
**options)
```

If auto_now is true, the date-time is updated whenever the object is updated. If auto_now_add is true, the date-time is set to the date when the object is created.

DecimalField: It is used to save data that has fixed decimal values like the price of any commodity (Rs. 10.99), length (1.990 km), etc. Example:

```
cost = models.DecimalField(max_digits=6, decimal_places=3,  
**options)
```

Here, max_digits is the maximum number of digits this field can store, including the digits after decimal and decimal_places is the number of digits after the decimal.

FloatField: It will store the float data type objects. Example:

```
percent_change = models.FloatField()
```

EmailField: It is just like the CharField with an email address check. If the value is not in a proper email ID format, it won't be saved in the database. Example:

```
email_id = EmailField(max_length= 100)
```

FileField: If you wish your application should accept uploads from users, you can use this field type. Example:

```
resume = FileField(upload_to=None, **options)
```

By default, it saves the files to the MEDIA_ROOT location. You can define your file location too.

ImageField: It is similar to FileField with a few extra options.

Example:

```
candidate_photo = ImageField(upload_to=None, height_field=None,  
width_field = None, max_length = 100)
```

IntegerField: It stores integers. The range is -2147483648 to 2147483647. Example:

```
roll_number = models.IntegerField()
```

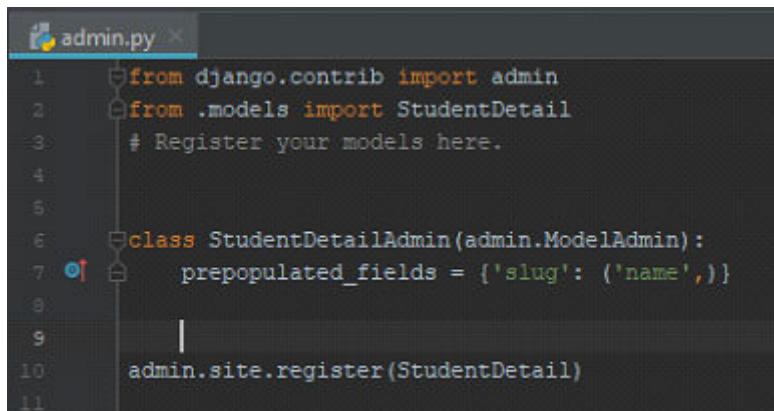
SlugField: This a very handy field generally used to identify objects stored in the database. It is similar to To implement this, you need to have a field in your model which can be used as a string containing hyphens, letters, numbers, etc. similar to an URL. For example:

```
models.py x  
1  from django.db import models  
2  from django.urls import reverse  
3  
4  
5  class StudentDetail(models.Model):  
6      full_name = models.CharField(max_length=200)  
7      dob = models.DateField()  
8      slug = models.SlugField(max_length=250, unique_for_date=dob)
```

Figure 7.2: SlugField

Slug for any article model with the title Democracy Under Threat will be In case of your project from the previous chapter, you can

have a SlugField by editing your models.py file and admin.py file of your student app as shown in [Figure 7.2](#) and [Figure](#) respectively:



```
1  from django.contrib import admin
2  from .models import StudentDetail
3  # Register your models here.
4
5
6  class StudentDetailAdmin(admin.ModelAdmin):
7      prepopulated_fields = {'slug': ('name',)}
8
9
10 admin.site.register(StudentDetail)
11
```

Figure 7.3: Setting SlugField in admin.py

In the admin.py file, you will need to set the slug field as shown in the preceding screenshot.

Here, if the name the full_name of the student is Ram then the corresponding slug will be These slugs are used to create URLs to specific objects. These can also be used to fetch absolute URLs in templates.

Relationship fields: These are used to set relationships between models like foreign keys in database tables. These are very important fields and hence are covered in detail later in this chapter.

These are the most commonly used fields. There are several other model fields that can come handy in special scenarios. So always

keep the Django's official documentation handy while coding so that you can refer to it quickly and get the best of the framework.

Field options

You have used these for a while now. Field options are nothing but the arguments of the model fields you have been passing all this while. Some field options are specific to a model field and some are generic and are available to all fields. Here, we will discuss the generic field options and their meaning.

blank: If this parameter is set to True, then the field can be blank. If it is set to False, the field cannot be blank. The default value is False. Example: -

```
full_name = models.CharField(max_length=200, blank=False)
```

null: If this parameter is set to True, then the empty values will be stored as null in the database. If it's set to the field cannot be null. The default value is The difference between blank and null is that blank represents an empty string and null represents No

choices: You have used this in your model. It needs to be defined first in an iterable consisting of another iterable with two elements. Then, this defined choice is passed as an argument in the model field.

Example:

```
std_choices = [('1', 'I'), ('2', 'II'), ('3', 'III'), ('4', 'IV'),
                ('5', 'V'), ('6', 'VI'), ('7', 'VII'),
                ('8', 'VIII'), ('9', 'IX'), ('10', 'X')]
std = models.CharField(choices=std_choices, max_length=1, default=None, null=True)
```

Figure 7.4: Choices option

The first element of the tuple is used as a reference and the second element is the value which will show up in the application.

db_column: This will be the name of the column in the database when you run makemigrations and migrate your model to the database. If it is not defined, Django uses the name of the field as the name of the column.

default: This is to set the default value of any field.

editable: The default value is If it's set to False, then that field will not be displayed at any place where it can be edited not even on the admin page.

help_text: This is the text with information about the field which you want to display to your users when they fill up any form.

Example:

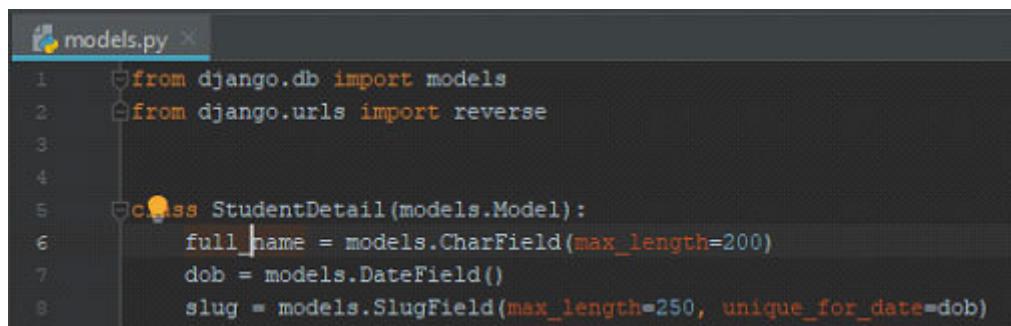
```
email_id = models.EmailField(max_length=50,
                           help_text='abc@xyz.com')
```

`primary_key`: If this set to the default auto-generated primary key `id` will not be created and that field will be used as the primary key. The default value is

`unique`: If this argument is Django runs a validation through the table and checks whether the value of the field is unique or not.

You have used this option earlier in the `SlugField` example. It contains a `DateField` or `DateTimeField` and is passed to a field which then will have to be unique for that date.

Example:



```
models.py
1  from django.db import models
2  from django.urls import reverse
3
4
5  class StudentDetail(models.Model):
6      full_name = models.CharField(max_length=200)
7      dob = models.DateField()
8      slug = models.SlugField(max_length=250, unique_for_date=dob)
```

Figure 7.5: `unique_for_date` option

Here, the students with the same `dob` must have a unique

`unique_for_month`: This is similar to The check is for a month instead of a date.

This is similar to The check is for the year instead of the date.

`verbose` This is a beautified version of the field name which is displayed UI. You can set a good name for your field. If it is not set, Django removes `_` from the field name and creates a

This is an advanced option. You can choose the validators you want to run on your field when any new object of the model class is created. Example: `EmailValidator`, etc. You write your validators and execute them on your fields to ensure that the correct data is stored in the database.

These were some commonly used fields. Again, Django has a lot of customizable field options. You can look for them if you need to work on some advanced projects. But most of the time, you will be using one of the preceding options. It is very helpful to have the model and field options handy while writing any Django model class.

Meta options

Meta data is a term you would hear very often in the computing world. It represents data about data. Let's say you have data in your table about students. This is your data. The information about how this is stored, what is the ordering, size of the table, etc. is the meta data for your data.

You already have used one of the meta options in your model earlier:

```
class Meta:  
    verbose_name_plural = "Student Details"
```

Figure 7.6: Sample meta option

The meta options are declared in the meta subclass of the model class. Here are some commonly used meta options:

app_label: If you are defining a model class outside the app and want to use it in your app, then just define the app_label to the name of the app you want to use it in. Example:

```
class Meta:  
    app_label = 'name_of_your_app'
```

`default_manager_name`: Every model has a model manager. A model manager is an interface via which Django interacts with the database. It helps in executing queries. The default model manager is `Objects` which you used while fetching the details of students from the table. You can have a custom model manager. In that case, this `meta` option comes handy. Define your model manager class (import it if defined in other files) and set `default_manager_name` to the manager you wish to use.

`db_table`: The default `db_table` name is If you want to set the name of the table for your model to a name other than the default, use this `meta` option. Example:

```
class Meta:  
    db_table = 'student_info'
```

`get_latest_by`: These are methods in the model manager like the `latest()`, `earliest()` which return the most recent objects. Using this `meta` option, you can set a model field based on which these methods will return the recent objects.

`managed`: This `meta` option decides if the model will be translated into a table in the database or not. By default, it's true; hence, when the migrations are run, a corresponding table is created or updated. If this option is set to false, the model will have no effect in DB.

`order_with_respect_to`: This is to set the ordering of objects based on a particular field.

ordering: When you retrieve data from DB using the view function in your views.py, you can set the order in which the objects are returned in the view function itself. Example:

```
def viewStudentList(request):
    students = StudentDetail.objects.all().order_by('first_name')
    return render(request, 'list.html', {'students': students})
```

Figure 7.7: Ordering in views

Now to set a default ordering for retrieval queries, you can use the ordering meta option and set to the field you want to use for ordering. You can have multiple fields here. Example:

```
class Meta:
    ordering = ['first_name', 'last_name']
    verbose_name_plural = "Student Details"
```

Figure 7.8: Setting ordering in meta options.

If you want to use the descending order, use - sign in front of the field name:

```
class Meta:
    ordering = ['-first_name', 'last_name']
    verbose_name_plural = "Student Details"
```

Figure 7.9: Reverse ordering

You can have multiple fields in order. If two objects have the same value for the field, then the next mentioned field will be used to order and so on.

`unique_together`: This is used to set the fields which must be unique in combination. Django implements this by adding the required unique constraints in the database attributes to make the fields unique together.

Example:

```
class Meta:  
    ordering = ['-first_name', 'last_name']  
    verbose_name_plural = "Student Details"  
    unique_together = ('first_name', 'father_name')
```

Figure 7.10: *unique_together* meta option

You can set several such pairs as shown in the following screenshot:

```
class Meta:  
    ordering = ['-first_name', 'last_name']  
    verbose_name_plural = "Student Details"  
    unique_together = (('first_name', 'father_name'), ('first_name', 'last_name'))
```

Figure 7.11: multiple pairs of *unique_together*

`verbose_name`: This is to set a name for the model which is easy to read. The Munged version of the class name is used by default like a model class name `StudentDetail` will become `student`

`verbose_name_plural`: You have used this in your project and it is used to define the plural name for the model class which is displayed on the UI. By default, Django adds an `s` at the end of the `Name`. This is sometimes incorrect or unwanted. For example, if you have a model class named `StudentData`, then Django will add an `s` at the end of the name on the UI and this will show up as `student data` which is incorrect. In such cases, it is better to define a

There are several more Meta options released with every new version release of Django. Check the official documentation for the latest updates.

Model methods

There are model managers and model methods. Model managers are classes implemented at the table level. The model methods are implemented at the row-level or instance/object level. You can define methods in your model class as you have been doing while programming in basic Python.

The model methods, also known as model instance methods, are functions defined in the scope of the model class like get_absolute_url method. You defined it in your model class and then invoked it in the template using the objects.

Example: You defined a model method in a model class:

```
def get_absolute_url(self):
    return reverse('student:viewStudentDetails', args=[self.student_id])
```

Figure 7.12: A sample of model option

Then, in you fetched a list of all the objects and passed it to the list.html template:

```
def viewStudentList(request):
    students = StudentDetail.objects.all()
    return render(request, 'list.html', {'students': students})
```

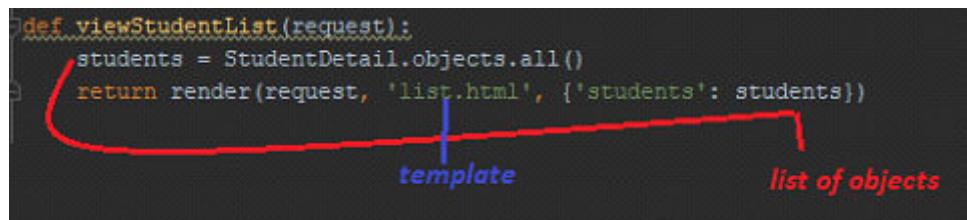


Figure 7.13: A view function using a template with the model method

Then, in the list.html template, you iterated over a list of objects received from the views function, picked one object at a time and used the object to invoke the model method:

```
list.html
1  {% extends "base.html" %}          object/instance
2  {% block title %}Our Institute{% endblock %}
3  {% block content %}
4      <h1>Institute Student Portal</h1>
5      <h2>Student List</h2>
6      {% for student in students %}    model method
7          <h2>
8              <a href="{{ student.get_absolute_url }}">
9                  {{ student.first_name }} {{ student.last_name }}
10             </a>
11         </h2>
12         <p class="date"> Enrollment Number {{ student.student_id }} </p>
13     {% endfor %}
14     {% endblock %}
```

Figure 7.14: A template invoking the model method.

This is how the methods defined in the model class can be used to solve a different purpose. It can be used for very basic needs like creating objects to advance uses like getting absolute URLs. Remember, these methods are invoked using the objects/instances of the model class. Thus, you can use them wherever you like the views.py or any other file. You just need to have the object there.

So, figure out your needs, define model methods, and use it accordingly.

Relationship between models

This is a very important concept. It is used to implement the relational database concept to avoid repetitive storage of data. These relationships between model classes are implemented using relationship model fields. There are three types of relationships namely, many-to-one, one-to-one, and many-to-many. To understand each of these in detail, we will need a problem statement. Let's consider an educational institution for which you are making an application to show details of students, teachers, and subjects. Let's break the requirements further:

A student can study many subjects and one subject can be studied by many students.

A teacher can teach only one subject (assuming teachers have only one specialization) but a subject can be taught by many teachers (assuming that the number of students is high, the institute has several teachers for a subject).

A teacher can teach many students and a student can attend classes of many teachers.

Also, we already have the StudetnDetails model, but we need to avoid unnecessary visibility of data in this table. So, we create another model for students (with data like teachers, subjects, etc.). Thus, we need to connect these two tables too. Now, for one object

of the Student table will be related to only one object of StudentDetail table and vis-e-vis.

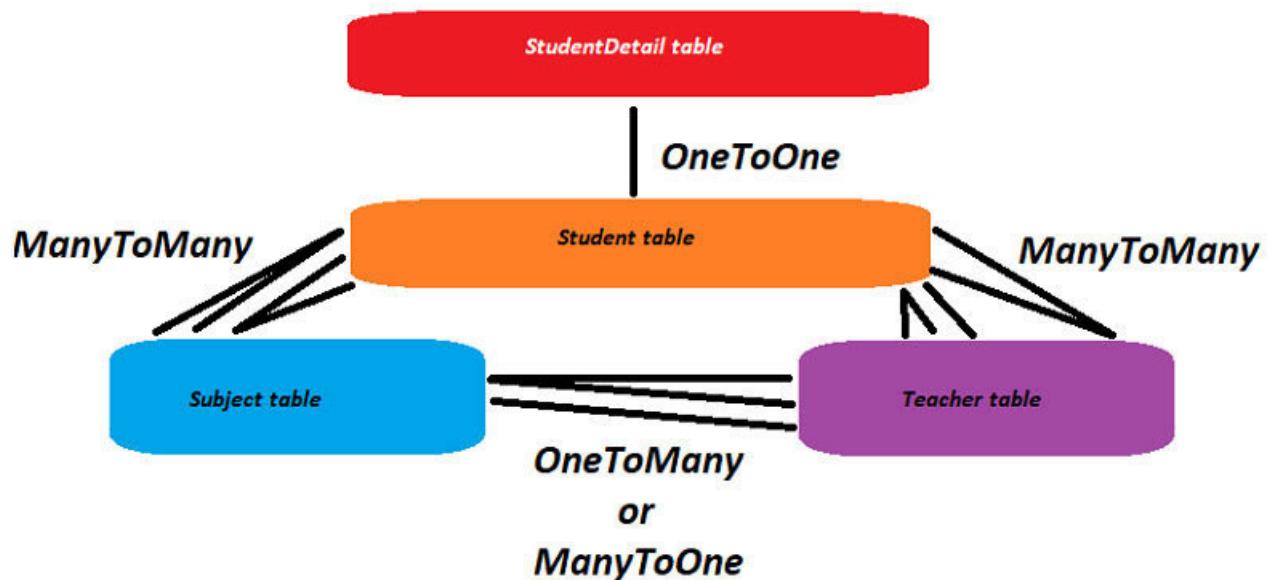


Figure 7.15: Relationship between models

To understand it better, you need to think about the number of relationships any object of one can have with the objects of other. Simplifying the preceding figure as follows:

An object of StudentDetail table can have a relationship with only one object of the Student table. Thus, it's a one-to-one relationship.

An object of the Student table can have a relationship with multiple objects of the Teacher table and vice-versa. Thus, it is a many-to-many relationship.

An object of the Student table can have a relationship with multiple objects of the Subject table and vice-versa. Thus, it is a many-to-many relationship.

An object of the Subject table can have a relationship with multiple objects of the Teacher table, but an object of the Teacher table can have a relationship with only one object of the Subject table. Thus, it is a one-to-many or many-to-one relationship.

We have established the relationships between our models. Let's create them. Let's take a look at the following code:

```
models.py
1  from django.db import models
2  from django.urls import reverse
3
4
5  class StudentDetail(models.Model):
6      gender_choices = [('M', 'Male'), ('F', 'Female')]
7      gender = models.CharField(choices=gender_choices, max_length=1,
8                                  default=None, null=True)
9      first_name = models.CharField(max_length=250)
10     last_name = models.CharField(max_length=250)
11     student_id = models.SlugField(max_length=250, unique=True)
12     fathers_name = models.CharField(max_length=250)
13     mothers_name = models.CharField(max_length=250)
14     dob = models.DateTimeField()
15     objects = models.Manager() # The default manager.
16     std_choices = [(1, 'I'), (2, 'II'), (3, 'III'), (4, 'IV'),
17                     (5, 'V'), (6, 'VI'), (7, 'VII'),
18                     (8, 'VIII'), (9, 'IX'), (10, 'X')]
19     std = models.CharField(choices=std_choices, max_length=1, default=None, null=True)
20
21     class Meta:
22         ordering = ['-first_name', 'last_name']
23         verbose_name_plural = "Student Details"
24
25     def get_absolute_url(self):
26         return reverse('student:viewStudentDetails', args=[self.student_id])
27
28
29     class Subject(models.Model):
30         name = models.CharField(max_length=50, unique=True)
31
32
33     class Teacher(models.Model):
34         name = models.CharField(max_length=50)
35         subject = models.ForeignKey('Subject', on_delete=models.CASCADE)
36
37
38     class Student(models.Model):
39         subject = models.ManyToManyField('Subject')
40         name = models.CharField(max_length=50)
41         student_id = models.OneToOneField('StudentDetail', on_delete=models.CASCADE)
42         teacher = models.ManyToManyField('Teacher')
```

Figure 7.16: A model class implementing different relationships

Create these three new model classes as shown in the preceding screenshot. This gives you an idea of how to implement different relationships. To find out what relationships to use in different scenarios, just find out the type of relationships between the objects of different

Note that these relationships are implemented by using a type of model field just like CharField, etc. Hence, these related fields will have specific field options. Let us discuss the options:

ManyToOne: This relationship is implemented by the ForeignKey field. The field options are as follows:

This option tells Django what to do if the source object is deleted. It is recommended to define this option always in the ForeignKey field.

`on_delete = models.CASCADE`: If the source object is deleted, then the related objects will also be deleted.

`on_delete = models.PROTECT`: This is used to protect the reference data from being deleted if the source object is deleted.

`on_delete=models.SET_NULL`: This is used to set the referenced object ForeignKey field to null if the source object is deleted.

`on_delete=models.SET_DEFAULT`: This is used to set the default value of the ForeignKey field if the source object is deleted.

`on_delete=models.SET(define_method)`: This option is used when you have any particular value or method defined which returns a value and you want that value to be saved in the ForeignKey field if the source object is deleted.

`on_delete=models.DO_NOTHING`: This is used when you do not want any action taken when the source object is deleted.

`limit_choices_to`: This sets the choices available for the value of the ForeignKey field in the referenced table.

Example:

```
subject = models.ForeignKey('Teacher', on_delete=models.CASCADE,
                           limit_choices_to={'maths': 'Maths', 'english': 'English'})
```

Figure 7.17: Implementing `limit_choices_to`

ManyToMany: This relationship is implemented by the ManyToMany field. To implement such a relationship between two models, Django creates another model in between them. This intermediate model is named using the field name in which the relationship is defined and the name of the other model. This model contains three fields: a primary key for every relationship, an object from the source table, and an object from the containing table.

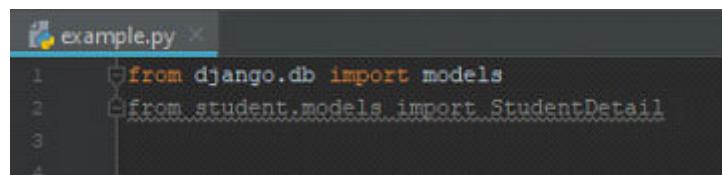
OneToOne: This relationship is implemented by the OneToOne field. It is similar to the ForeignKey field with It is mostly used to connect two tables with information about similar things that cannot be stored at one table for certain reasons.

These were the relationships between models. This is just like a link table in the database which has connected data so that you do not have all the data in all the tables. Think of some scenarios and try writing model relationships.

Connecting models

There may be scenarios where you have to use the same model class in different apps or situations in which you have to create a model class outside your app. This is not a problem. Example:

```
from appName.models import
```



```
example.py
1 from django.db import models
2 from student.models import StudentDetail
3
4
```

Figure 7.18: Importing models from elsewhere

You just need to import the model class and use it as it's written in your models.py. It is easy. Here, the student is the app name from which you want to import the model, .models indicates the file name of the student app in which the model class is written and StudentDetail is the model class which has to be imported.

Conclusion

It has been a long chapter. You studied models in its fields, field options. Models are classes and their attributes are fields. In terms of databases, classes are tables and fields are columns. The field options are column properties. The Meta class is relevant to table properties. It is not expected of you to remember all these. You will get a hang of it when you start coding. For now, you must understand the concept of models in Django. You must be able to figure out the type of field, field option, meta option, and relationships you must choose or look for while creating models. Now, you would be able to create your models for any given scenario. Do browse through the official Django documentation about models to look for more specific code and latest updates. In the next chapter, you will read about url.py and

In the next chapter, you will read about the views of Django. Views are the classes that use the models and fetch data from the tables or insert data into the tables. You will read how views interact with models to fetch and insert data and how to render a template in response to a request.

Questions

What are the model fields?

Which model field is used to store date and time?

What is the use of

Which model field options are used to establish relationships between two models?

Why is the model relationship the same as the foreign key in SQL tables?

CHAPTER 8

Django Views

A view can be a standalone function or a class that has methods that can be invoked using its objects, or you can use one of the many built-in views provided by Django. You will read about this in this chapter.

You have read about models; now, you should learn about the component which is going to use the models to interact with the database - the views. Views are the core of any Django based application. Views are the layers where most of the action of the application takes place. This is the component you will use to control the behavior of the application.

Structure

Introduction

Types of views

When to use class and function-based views

Built-in class-based views

Conclusion

Objectives

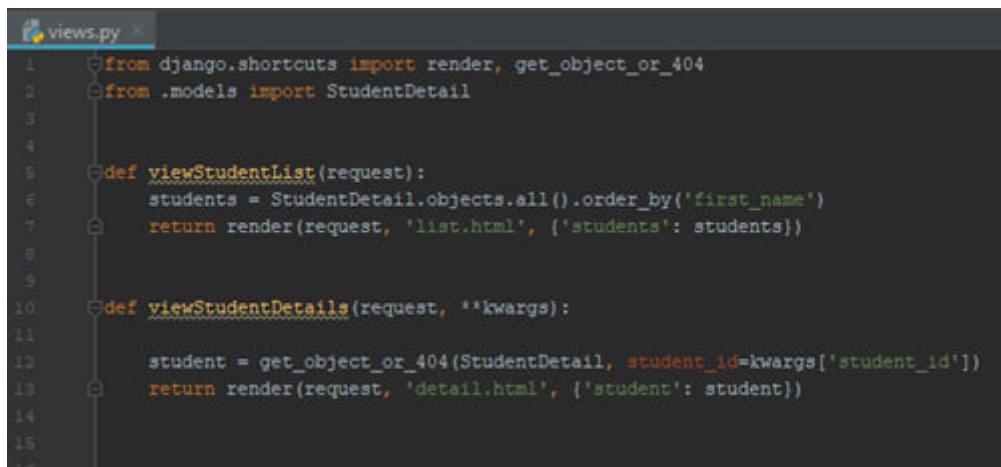
Understand the concept of views.

Understand the types of views.

Understanding the usage of different types of views.

Introduction

A view in a Django project is the layer of architecture where the logic is coded. It is invoked when the URL of the request is matched with a url-pattern present in the urls.py file. Once invoked, a view executes the logic written which generally includes interacting with the database to fetch/insert/update any data and then responding with a template and context data to be rendered on the page after executing the logic. The following screenshot displays the views.py file you have worked with in your project from the previous chapter and see what you have used already:



```
views.py
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3
4
5  def viewStudentList(request):
6      students = StudentDetail.objects.all().order_by('first_name')
7      return render(request, 'list.html', {'students': students})
8
9
10 def viewStudentDetails(request, **kwargs):
11
12     student = get_object_or_404(StudentDetail, student_id=kwargs['student_id'])
13     return render(request, 'detail.html', {'student': student})
14
15
```

Figure 8.1: Django Function-based views

At a glance, one can say that the views.py file may contain import statements and view functions. Further, the view functions accept an HTTP request to do some processing and render an HTML template in response to the request. This is a very basic structure

of a views.py file. It can contain several view functions or classes. Further, those classes can have several methods and subclasses. Let's take a look at it in more detail.

Types of views

Based on the definition, there are two types of views: function-based and class-based.

Function-based views

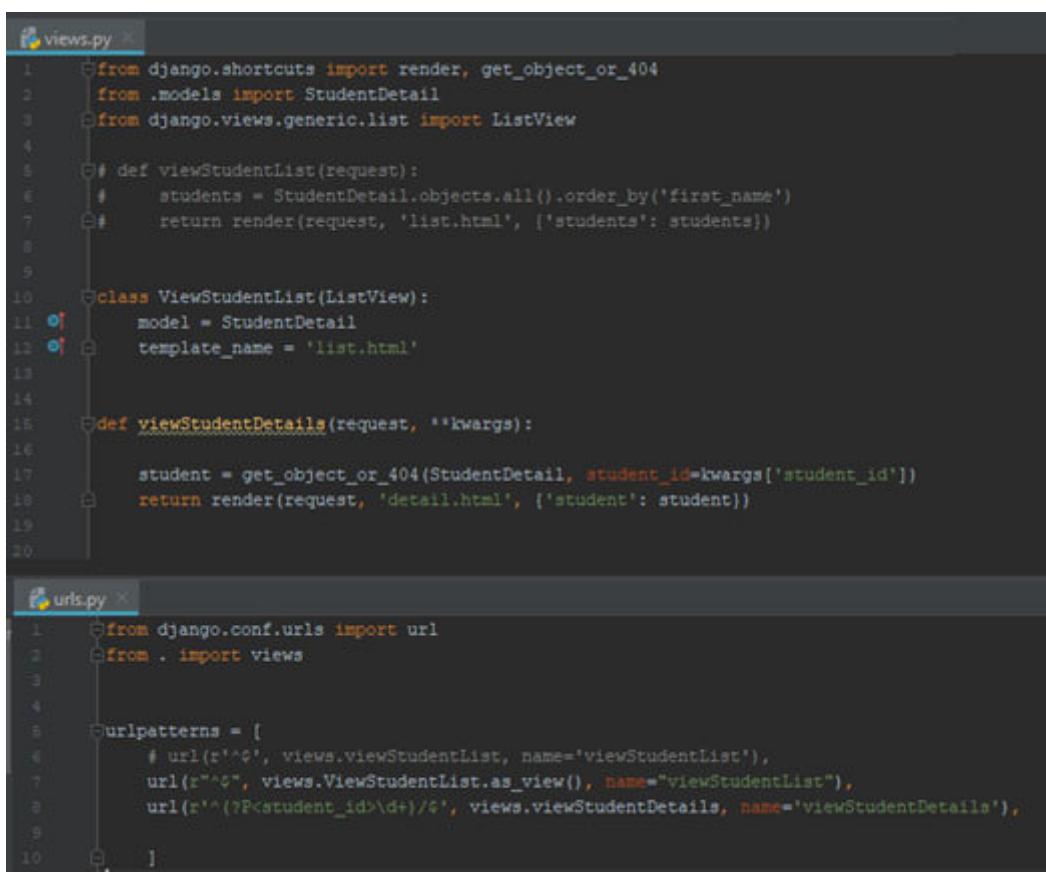
You have already created and worked on a function-based view. A function-based view is a function that gets invoked from a URL pattern. It takes the request argument and some other optional argument. Based on the arguments, it performs the steps mentioned in the function. Lastly, it returns a template/HTML/response to the request received initially. This view should be used if you have some specific logic or steps to perform for the requests.

Class-based views

Earlier, only function-based views existed. Later, programmers realized that they were writing similar view functions in several projects like list views, detail views, registration form views, etc. Hence, classes-based views were created.

Even class-based views are implemented by a function only. You will see that when we implement these.

Example: Let's implement the viewStudentList view function via class-based views:



The screenshot shows a code editor with two files open:

- views.py**: This file contains Python code for defining views. It includes imports for django.shortcuts, django.models, and django.views.generic.list. It defines a function-based view `viewStudentList` and a class-based view `ViewStudentList` (which inherits from `ListView`). The class-based view specifies the model as `StudentDetail` and the template name as `'list.html'`. It also defines a function-based view `viewStudentDetails` which uses `get_object_or_404` to retrieve a student by their ID and renders a detail page.
- urls.py**: This file contains Python code for defining URLs. It imports url from django.conf.urls and views from the current directory. It defines a URL pattern `url(r'^$', views.viewStudentList, name='viewStudentList')`, another for the class-based view `url(r'^$', views.ViewStudentList.as_view(), name="viewStudentList")`, and a third for the function-based detail view `url(r'^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails')`.

Figure 8.2: Django class-based views.py and urls.py

When you realize that a view can be utilized for several purposes by making minimum changes in the attribute of the class, class-based views should be used. Django provides a big list of built-in class-based views. These are available in classes that you can inherit in your class-based view class and use as per your requirements.

When to use class and function-based views

You need to make a choice before writing your views and URL patterns. Both class and function-based views have their use case scenarios. Function-based views are classics and can do the work anytime whereas class-based views come with reduced coding efforts. So, let's see when you can use class-based views. Follow the given questionnaire and you will find out if class-based views can be used:

Is there a built-in class-based view which some-what matches your requirements?

Can you use any class-based view by modifying its attributes?

Can you meet your requirements by creating a sub-class of a class-based view without modifying the in-built framework code of the view?

If the answer to any of the questions is Yes, you can use class-based views. Otherwise, you can use function-based views. Although, it largely depends on the developer's choice. This might not make much sense now to some of the readers. Continue with the chapter and refer to it later when you have a better idea of class-based views. It will make much sense then.

[Built-in class-based views](#)

Django offers a big set of built-in class-based views. There are several categories. Two of them are base views and generic views.

Base view

When your requirements do not match to any of the built-in generic class-based views and you still want to use class-based views, you will use base views. This provides an empty structure of class-based views, and you can give it the shape as per your requirements. These are further implemented by inheriting three classes: View, and

View

This is the most basic class-based view. All other class-based views inherit from this class-based view.

Example: Here, we have imported the class View from Then, we created our view class HelloWorld which inherited the imported View class. Further, we defined a get method that takes a few optional arguments and returns an HttpResponse to the request. You will note that the get method is similar to the view-functions we used to define initially.

Further, in just add an URL pattern as shown in the following screenshot. This is a standard way for writing URL patterns corresponding to a class-based view:

The screenshot shows two Python files open in PyCharm: `views.py` and `urls.py`.

views.py:

```
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3
4  from django.http import HttpResponseRedirect
5  from django.views import View
6
7
8  class HelloWorld(View):
9
10     def get(self, request, *args, **kwargs):
11         return HttpResponseRedirect('Hello, World!')
12
13
14     def viewStudentList(request):
15         students = StudentDetail.objects.all().order_by('first_name')
16         return render(request, 'list.html', {'students': students})
17
```

urls.py:

```
1  from django.conf.urls import url
2  from . import views
3
4
5  urlpatterns = [
6      url(r'^$', views.viewStudentList, name='viewStudentList'),
7      url(r'^mine/$', views.HelloWorld.as_view(), name='hello-world'),
8      url(r'^^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
9  ]
```

Figure 8.3: Django class-based `views.py` and `urls.py` files

What's happening in the code shown in the preceding screenshot? If you wish to see the code of the `View` class imported here, in PyCharm, open your `views.py` file where you have written the preceding code, press `Ctrl`, and hover over `View` in line number 8 of the `views.py` file, as shown in the preceding screenshot. It will be highlighted. Click on it and the `View` class definition will open. This is how you can open definitions of any class, function, etc.

Since the `HelloWorld` class has inherited the `View` all the components of the `View` class will be accessible to the `HelloWorld` class. In the `View` class definition, the following components are defined:

```
http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head',  
'options', 'trace']
```

These are the http request methods, which are supported by the view. If any other method is received in the request, it is not executed.

If you check the code of the View class, you will see the following methods defined. You can use these methods to customize your class-based view:

`as_view()`: This method is the view function of the class-based view. It is defined in every class-based view. This method is called when the URL matches the url-pattern defined in the urls.py file. Check line number 7 of the urls.py file in the preceding screenshot.

`setup(self, request, *args, **kwargs)`: This method initializes the variables like `request` (which needs to be the server), optional and `kwargs` which are received along with the request like `student_id` in case of the `viewStudentDetail` of url-pattern of the project you worked on earlier.

`dispatch(self, request, *args, **kwargs)`: This method accepts the request, checks the type of request like `get`, `post`, etc. and delegates it to the corresponding `get()` or `post()` method accordingly.

`http_method_not_allowed(self, request, *args, **kwargs)`: If the method in the request is other than the methods mentioned in this method is called.

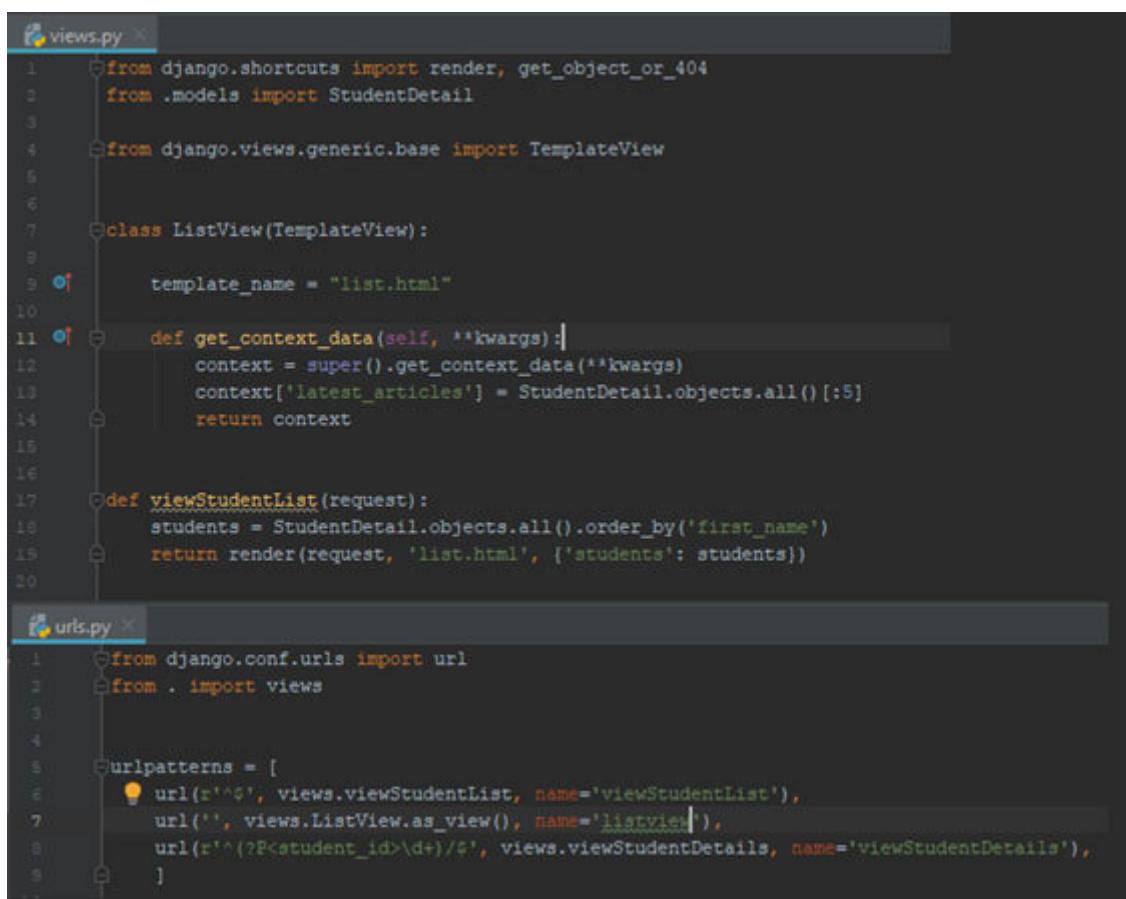
`options(self, request, *args, **kwargs)`: This method is responsible for responding to the views.

But you don't have to worry about these. You can simply inherit the base class to your View class and use accordingly.

Templateview

This view can be used when you have any template to render in response to the HTML request.

Example: In this view, we set up an attribute `template_name` which is to be rendered in response to the request. Further, we define a context that will be passed to the template. It is very much similar to the `viewStudentList` function:



The screenshot shows two Python files in a code editor. The top file is `views.py` and the bottom file is `urls.py`.

views.py:

```
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3
4  from django.views.generic.base import TemplateView
5
6
7  class ListView(TemplateView):
8
9      template_name = "list.html"
10
11     def get_context_data(self, **kwargs):
12         context = super().get_context_data(**kwargs)
13         context['latest_articles'] = StudentDetail.objects.all()[:5]
14         return context
15
16
17     def viewStudentList(request):
18         students = StudentDetail.objects.all().order_by('first_name')
19         return render(request, 'list.html', {'students': students})
20
```

urls.py:

```
1  from django.conf.urls import url
2  from . import views
3
4
5  urlpatterns = [
6      url(r'^$', views.viewStudentList, name='viewStudentList'),
7      url('', views.ListView.as_view(), name='listview'),
8      url(r'^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
9  ]
```

Figure 8.4: Django class-basedviews.py and urls.py files

Here, you have the views.py file and the urls.py file for implementing the ListView class-based view. Notice the difference and similarity. When you use a function-based view, you put the name of the function which is to be invoked in the When you use the class-based view, you put the as_view() method of the ListView in the url-patterns. This is because the url-pattern needs a callable object there; hence, we use the as_view() method to invoke the class-based view methods.

RedirectView

This view is used to redirect to another URL. Suppose you receive a request in which you need the user to land on another page, which has a different URL, you can use this view. You can redirect to another website or another url-pattern from your urls.py file, etc.

In the redirect view, we have the following attributes:

url: The URL to which you want to set up a redirect.

pattern_name: Name of the URL pattern you want to redirect to.

permanent: This checks whether the redirect is permanent or not (True/False). The default value is False. In the case True, the HTTP response status code of redirect is 301 and in case of False, the status code is 302.

query_string: A query_string is received with the request. If this attribute is set to True, the received query_string will be passed to the new URL formed. If the query_string will not be passed. The default value is False.

Methods in

`get_redirect_url(*args, **kwargs)`: This method is responsible for creating the URL for redirection. The arguments here are obtained from url-patterns of the request. The default implementation uses the URL as a starting string and performs expansion of % named parameters in that string using the named groups captured in the URL.

If the URL is not set, `get_redirect_url()` tries to reverse the `pattern_name` using what was captured in the URL (both named and unnamed groups are used). If requested by it will also append the query string to the generated URL. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

Example:

The screenshot shows a code editor with two tabs open: `views.py` and `urls.py`.

`views.py` content:

```
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3
4  from django.views.generic.base import TemplateView, RedirectView
5
6
7  class StudentRedirectView(RedirectView):
8
9      permanent = False
10     query_string = True
11     pattern_name = 'student-detail'
12
13     def get_redirect_url(self, *args, **kwargs):
14         student = get_object_or_404(StudentDetail, pk=kwargs['pk'])
15         student.update_counter()
16         return super().get_redirect_url(*args, **kwargs)
17
```

`urls.py` content:

```
1  from django.conf.urls import url
2  from . import views
3  from django.views.generic.base import RedirectView
4
5  urlpatterns = [
6      url(r'^$', views.viewStudentList, name='viewStudentList'),
7      url(r'', views.ListView.as_view(), name='listview'),
8      url(r'^counter//', views.StudentRedirectView.as_view(), name='student-counter'),
9      url(r'^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
10     url(r'^go-to-django/$', RedirectView.as_view(url='https://djangoproject.com'), name='go-to-django'),
11 ]
12
```

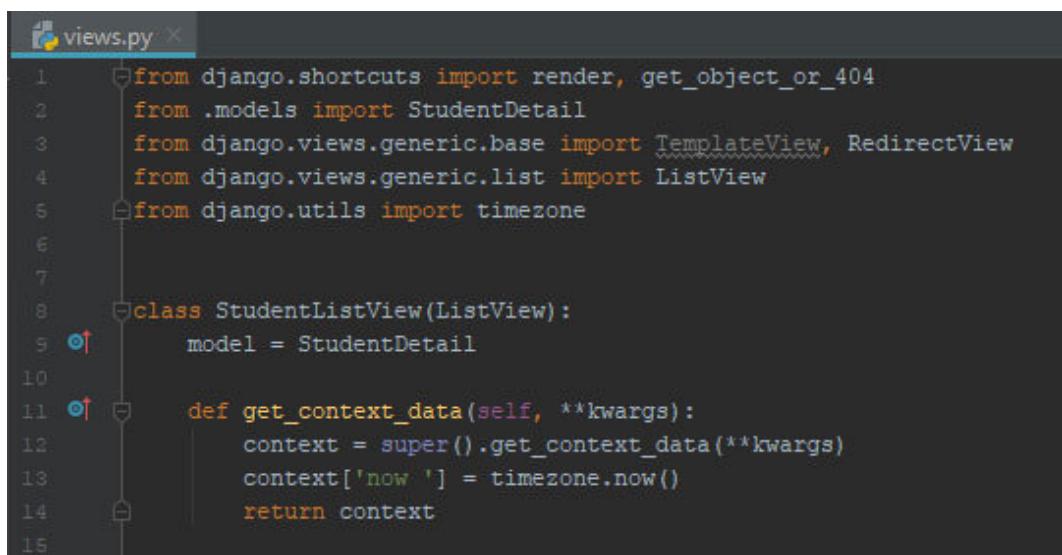
Figure 8.5: Django class-based views.py files and urls.py

Generic views

There are generic views available to display the data List and Details view. You have implemented both of these using function-based views in the project. Let's see this implementation in class-based views.

List view

This view shows a list of all the objects. For example: The following screenshot displays the views.py file with the StudentListView class-based generic list view:



```
views.py
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3  from django.views.generic.base import TemplateView, RedirectView
4  from django.views.generic.list import ListView
5  from django.utils import timezone
6
7
8  class StudentListView(ListView):
9      model = StudentDetail
10
11     def get_context_data(self, **kwargs):
12         context = super().get_context_data(**kwargs)
13         context['now'] = timezone.now()
14         return context
15
```

Figure 8.6: Django class-based views.py

This is the generic It implements the query set self.object_list which usually will contain a list of all the objects. Thus, you need to update your template as per the generic view as the view passes an object_list with the list of objects:

```
list.html
1  {% extends "base.html" %} 
2  {% block title %}Our Institute{% endblock %} 
3  {% block content %} 
4      <h1>Institute Student Portal</h1> 
5      <h2>Student List</h2> 
6      <ul> 
7          {% for item in object_list %} 
8              <li> 
9                  <h2><a href="{{ student.get_absolute_url }}> {{ student.first_name }} {{ student.last_name }} </a></h2> 
10                 <p class="date"> Enrollment Number {{ student.student_id }} </p> 
11             </li> 
12         {% empty %} 
13             <li>No Students yet.</li> 
14         {% endfor %} 
15     </ul> 
16  {% endblock %}
```

Figure 8.7: Django Function-based views

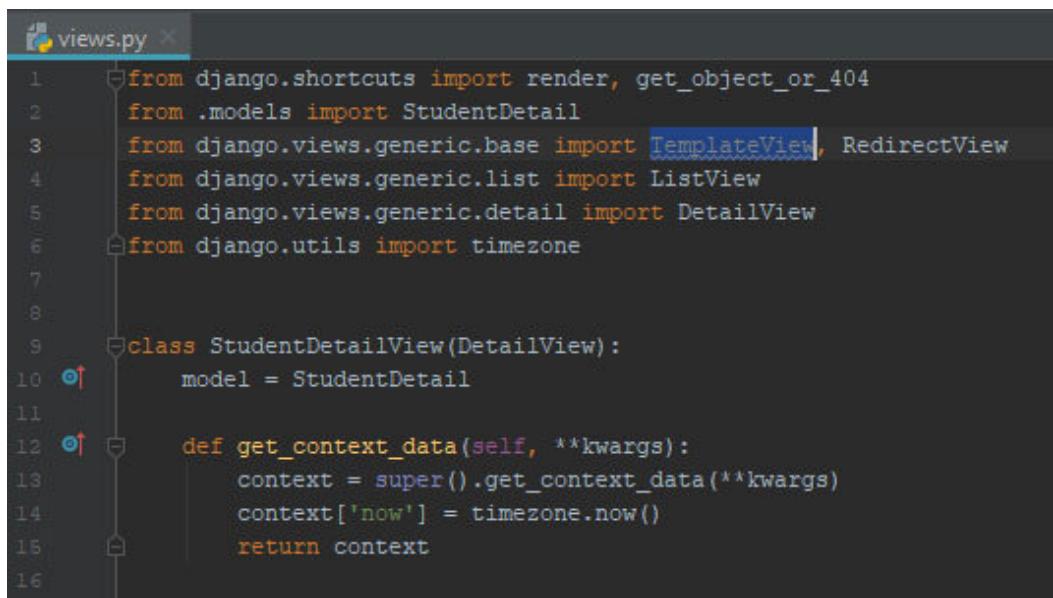
Now, once the view and template are ready, you need to work on the urls.py file and update the url-patterns with the view you created. The following screenshot displays the urls.py file. Make your url-pattern look like the following screenshot:

```
urls.py
1  from django.conf.urls import url 
2  from . import views 
3  from django.views.generic.base import RedirectView 
4 
5  urlpatterns = [ 
6      # url(r'^$', views.viewStudentList, name='viewStudentList'), 
7      url('', views.StudentListView.as_view(), name='listview'), 
8      url(r'^(?P<student_id>\d+)$', views.viewStudentDetails, name='viewStudentDetails'), 
9      url(r'counter/<int:pk>/', views.StudentRedirectView.as_view(), name='student-counter'), 
10     url(r'go-to-django/', RedirectView.as_view(url='https://djangoproject.com'), name='go-to-django') 
11 ]
```

Figure 8.8: Django urls.py file for redirect view

Detail view

This view receives an argument along with a request which has an object identifier and displays the data of that object. For example: The following screenshot displays the views.py file with the StudentDetailView class-based generic detail view:



```
views.py
1  from django.shortcuts import render, get_object_or_404
2  from .models import StudentDetail
3  from django.views.generic.base import TemplateView, RedirectView
4  from django.views.generic.list import ListView
5  from django.views.generic.detail import DetailView
6  from django.utils import timezone
7
8
9  class StudentDetailView(DetailView):
10     model = StudentDetail
11
12     def get_context_data(self, **kwargs):
13         context = super().get_context_data(**kwargs)
14         context['now'] = timezone.now()
15         return context
16
```

Figure 8.9: Django class-based detail view

This is the generic It implements the query set self.object which usually will contain data of the Thus, you need to update your template as per the generic view as the view passes an object with the data of The following screenshot displays your updated template:

```
1  {% extends "base.html" %}          2  {% block title %} Student Details {% endblock %} 3  {% block content %} 4      <h1> 5          <a href="/">Institute Student Portal</a> 6      </h1> 7      <h2>Student Details</h2> 8      <ul> 9          <li>Student Name: {{ object.first_name }} {{ object.last_name }}</li>10         <li>Enrollment Id: {{ object.student_id }}</li>11         <li>Father's Name: {{ object.fathers_name }}</li>12         <li>Mother's Name: {{ object.mothers_name }}</li>13         <li>Date of Birth: {{ object.dob }}</li>14         <li>SID: {{ object.std }}</li>15         <li>Time: {{ object.time }}</li>16     </ul>17  {% endblock %}
```

Figure 8.10: Django detail view template

Again, we need to work on the urls.py file and update the url-patterns with the view you created. The following screenshot displays the urls.py file. Make your url-patterns look like the following screenshot:

```
1  from django.conf.urls import url
2  from . import views
3  from django.views.generic.base import RedirectView
4
5  urlpatterns = [
6      # url(r'^$', views.viewStudentList, name='viewStudentList'),
7      url('', views.StudentListView.as_view(), name='listview'),
8      # url(r'^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
9      url(r'^(?P<student_id>\d+)/$', views.StudentDetailView.as_view(), name='viewStudentDetails'),
10     url(r'counter/<int:pk>/', views.StudentRedirectView.as_view(), name='student-counter'),
11     url(r'go-to-django/', RedirectView.as_view(url='https://djangoproject.com'), name='go-to-django')
12 ]
```

Figure 8.11: Django urls.py file for detail view

These were some common class-based built-in views that you will be using a lot in your projects. These built-in views cover almost

every scenario. You can customize them. You just need to inherit the view which you want to use and then go through its code in the library. Find out the method which you need to modify and re-define that method in your view class definition. That will override the default method and your method will be executed.

Conclusion

You read about the different types of views – class-based and function-based views. Your project from the previous chapters used function-based views. You now know which view to choose. If you have some logic that is not present in the built-in views or you have to do a lot of customization in the built-in view, then you need to create your views. Now, if the view is to be used only once, use the function-based view. I see that you can re-use it at some places to create a class-based view. You have already created a function-based view and now you can also work with class-based views. You can choose from the different options available and create a view as per the requirement of your project. You can browse through the official Django documentation to read more about built-in views and how they involve templates.

In the next chapter, you will read about templates. You must be confused about the code being written in the templates. Some look like HTML and some resemble Python. In the next chapter, you will read about this particular language called Django templating language. It is an HTML that can have the Python code enclosed in tags.

Questions

What are class-based views?

What are the different built-in class views?

What is the difference between generic and base class-based views?

When is it recommended to use a function-based view?

When is it recommended to use a class-based view?

How do we customize any built class-based view?

CHAPTER 9

Django Templates

The web pages you saw in your project earlier were rendered by HTML code present in the templates. The **Django Templating Language** is used to write templates. These templates are responsible for incorporating the dynamic data into the webpages.

If you want a dynamic website where data is displayed, it is important that you read about the Django Templating Language. It provides a lot of tags and features using which you can write the Python code in your HTML files.

In this chapter, you will learn more about the templates of Django. You created a template and used it in your project to display and respond. Django provides an inbuilt template engine that we have been using. Here, you will read about how to write templates and their configuration.

Structure

Introduction

Configuration

Django Templating Language

Debugging

Summary

Objectives

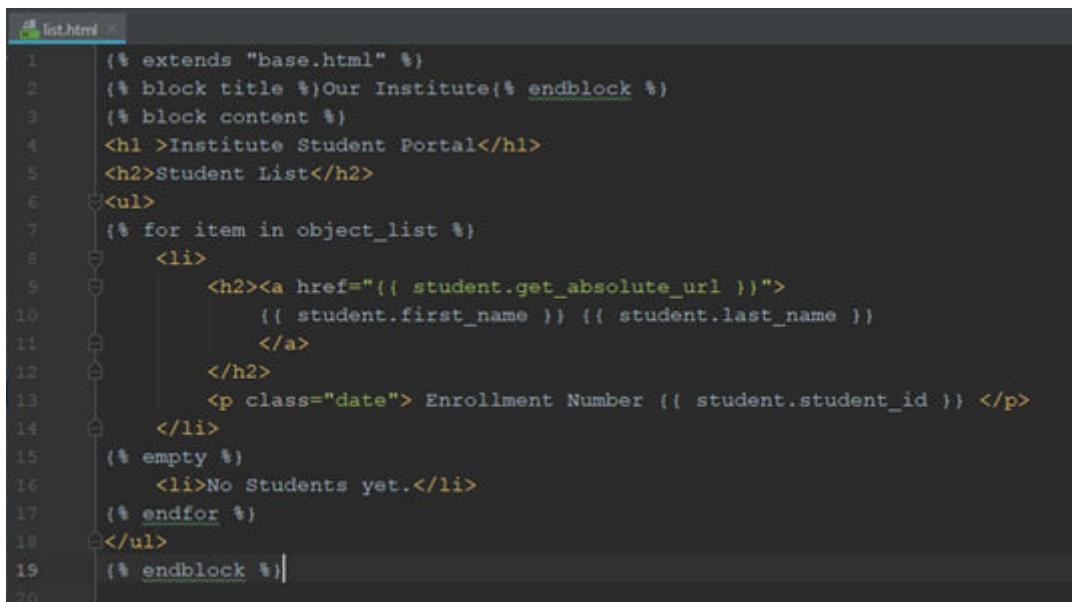
Understanding the template component of the Django architecture.

Understanding the different tags used in templates.

Understanding reusability of templates – Template Inheritance.

Introduction

Let's pull out the list.html file from your project and see what has been used already. You will get an idea about the code from the following screenshot. Here, everything that is written inside '{' and '}' is the Python code. This called the **Django Templating**. Everything else is HTML. The following screenshot displays a sample template:



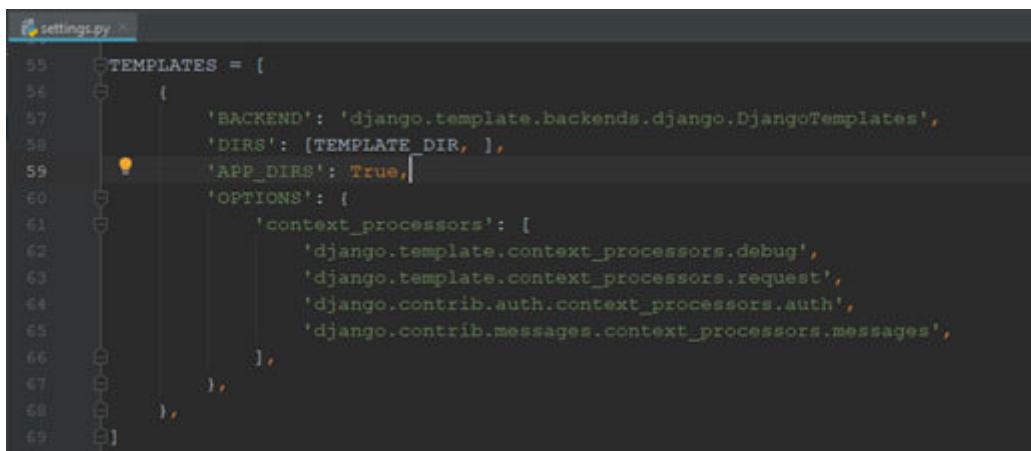
```
list.html
1  {% extends "base.html" %} 
2  {% block title %}Our Institute{% endblock %}
3  {% block content %}
4      <h1>Institute Student Portal</h1>
5      <h2>Student List</h2>
6      <ul>
7          {% for item in object_list %}
8              <li>
9                  <h2><a href="{{ student.get_absolute_url }}>
10                     {{ student.first_name }} {{ student.last_name }}
11                 </a>
12             </h2>
13             <p class="date"> Enrollment Number {{ student.student_id }} </p>
14         </li>
15     {% empty %}
16     <li>No Students yet.</li>
17   {% endfor %}
18   </ul>
19   {% endblock %}
```

Figure 9.1: Django Template

We will read more about the templating language in this chapter. There are other templating languages that are supported by Django like Jinja2 and Mako. You can also configure the template engines and use them in your projects.

Configuration

To use any of the templating languages, you need to configure its engine in the settings.py file. This is done in the TEMPLATES tag. By default, this is empty. Go to settings.py file of your project and check for the TEMPLATE tag:



```
settings.py
55     TEMPLATES = [
56         {
57             'BACKEND': 'django.template.backends.django.DjangoTemplates',
58             'DIRS': [TEMPLATE_DIR, ],
59             'APP_DIRS': True,
60             'OPTIONS': {
61                 'context_processors': [
62                     'django.template.context_processors.debug',
63                     'django.template.context_processors.request',
64                     'django.contrib.auth.context_processors.auth',
65                     'django.contrib.messages.context_processors.messages',
66                 ],
67             },
68         },
69     ]
```

Figure 9.2: Settings.py file for templates

Here, you see a dictionary with keys as setting and values as configuration for the setting. Let's take a look at them one by one:

BACKEND: This is the path to the template engine class which has the API for Django's template engine. Since we are using the Django Templating Language, the path is to the DjangoTemplates class. In case of Jinja2, the path would be Of course, you would need to install Jinja2 first.

DIRS: This is the location where the engine would look for templates. Your earlier project's template directory in app/templates had only one app so we defined that location for templates in the settings.py file.

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'student/templates')
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR, ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Figure 9.3: Settings.py file for templates

You need to follow a more generalized structure as you cannot define a separate template folder for all apps in your project. Generally, the folder structure for templates is in For example, if the name of your app is account, then its templates would be in the account/template/accounts directory.

In this case, the template directory would look like the following screenshot:

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR, ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Figure 9.4: *Settings.py file for templates*

APP_DIRS: If this is set to true, the engine will look for a template inside the individual app directories as mentioned earlier.

OPTIONS: This is a dictionary with advanced settings about the template backends. It includes settings like libraries, loaders, etc.

You will not need to worry about these settings until you do very advanced projects. It's not necessary to remember all the settings, but it's important to remember where to look for these settings and how to follow the official Django documentation.

Template inheritance

Template inheritance is a concept that allows us to inherit the contents of other templates to child templates. Just like how you inherited base.html to Every website has some stuff that is common to all the pages like styling, header, footer, etc. Thus, inheriting the common stuff to the page's template will help in avoiding rewriting the same HTML code for every page. This is done by using the extends tag.

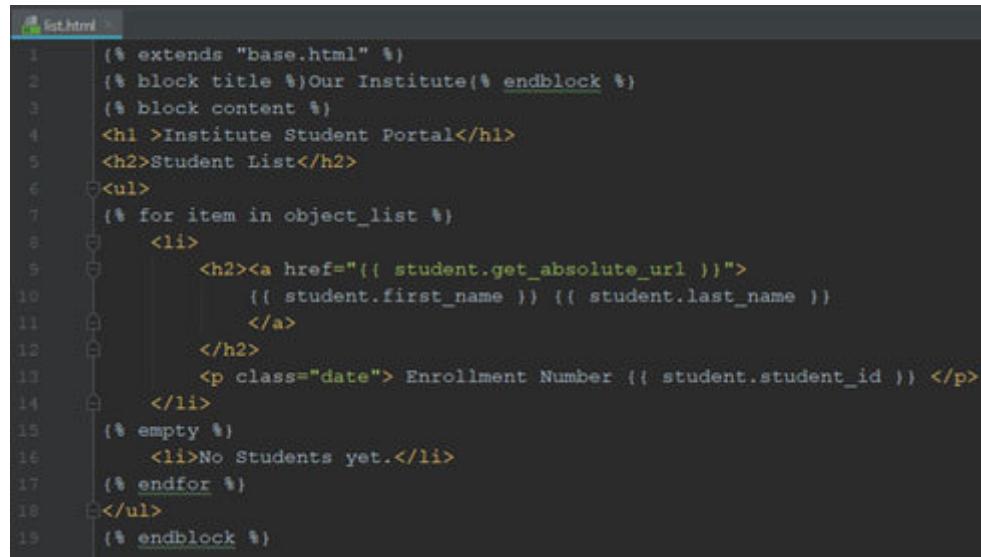
Example: Here, in you will see the tags block content and endblock In list.html also, you will see the tags block content and endblock

```
<head>
    <title>Institute Student Portal</title>
    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
        integrity="sha384-BVYii3IF8eOqCI4QQuFjG�A7HdYR8iZJxPlh+I7JYd+7R7hB9lW&lt;!-->
        crossorigin="anonymous">
<body class="loader">
    <div class="content container">
        <div class="row">
            <div class="col-md-8">
                <div class="studentdetails">
                    (block content)
                (endblock)
            </div>
        </div>
    </div>
    (# SCRIPTS)
    <script type="text/javascript" src="{% static 'js/student.js' %}"></script>
</body>
</html>
```

Figure 9.5: Template Inheritance – parent template

Now like any other inheritance, the child – list.html will have its properties and parent's properties. If there is a property present in

both child and parent, the property of the child will prevail. The following screenshot displays the child template which will inherit the preceding parent template:



```
list.html
1  {% extends "base.html" %} 
2  {% block title %}Our Institute{% endblock %} 
3  {% block content %} 
4      <h1>Institute Student Portal</h1> 
5      <h2>Student List</h2> 
6      <ul> 
7          {% for item in object_list %} 
8              <li> 
9                  <h2><a href="{{ student.get_absolute_url }}"> 
10                     {{ student.first_name }} {{ student.last_name }} 
11                 </a> 
12                 </h2> 
13                 <p class="date"> Enrollment Number {{ student.student_id }} </p> 
14             </li> 
15         {% empty %} 
16             <li>No Students yet.</li> 
17         {% endfor %} 
18     </ul> 
19  {% endblock %}
```

Figure 9.6: Template Inheritance – child template

Similarly, when list.html inherited base.html with the command extends “base.html” it inherited all the code of base.html. Since, both have tags block content and endblock the tag of the child will prevail. This is called **template**

In other words, the base.html is a skeleton and the child templates will fill the base templates with data using the block tags.

Django Templating Language

Just like any markup language, the Django tTemplating Language is also a markup language with Python strings. The Python strings, which are the Python code, that are written along with HTML inside certain tags when executed by the engine, create dynamic HTML pages as a response.

Syntax

The structure of statements to write the Python code in the text/HTML files is as follows:

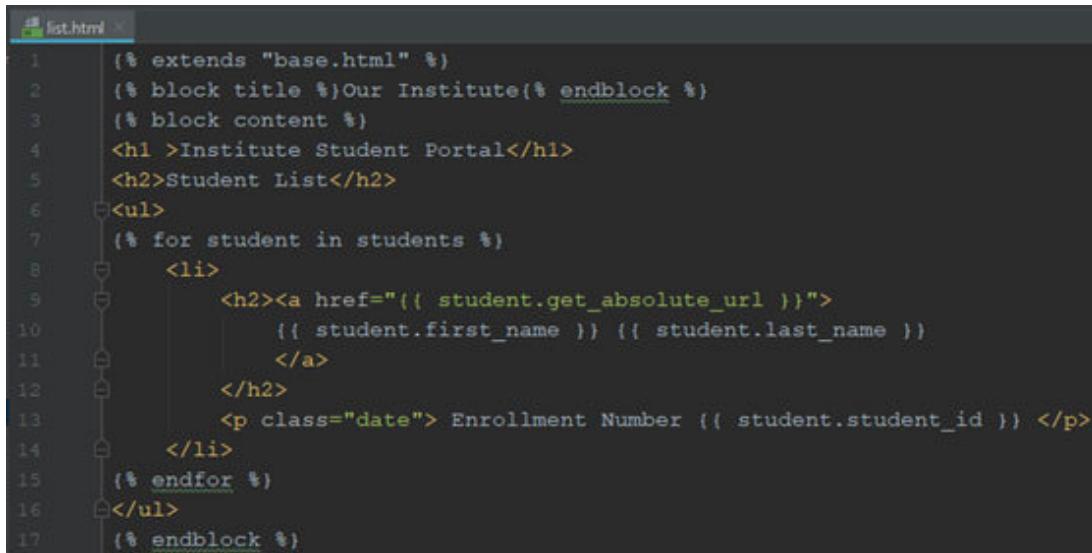
Variables:

The variables return a value from the context received from the views. A context is a dictionary with keys as variable_name and values as In the following view, the HTML file returned is list.html. The context is the dictionary passed as a parameter, where students are variable_name present in the list.html and students is the list of objects obtained from the query set in the view:

```
def viewStudentList(request):
    students = StudentDetail.objects.all().order_by('first_name')
    return render(request, 'list.html', {'students': students})
```

Figure 9.7: views.py

When the following template is executed, it receives the context, {'students':student}, and replaces the variables {{student.get_absolute_url}}, {{student.first_name}}, and {{student.student_id}} with the values obtained after processing the list of students:



```
list.html
1  {% extends "base.html" %} 
2  {% block title %}Our Institute{% endblock %} 
3  {% block content %} 
4  <h1>Institute Student Portal</h1> 
5  <h2>Student List</h2> 
6  <ul> 
7  {% for student in students %} 
8  <li> 
9  <h2><a href="{{ student.get_absolute_url }}> 
10    {{ student.first_name }} {{ student.last_name }} 
11    </a> 
12  </h2> 
13  <p class="date"> Enrollment Number {{ student.student_id }} </p> 
14 </li> 
15 {% endfor %} 
16 </ul> 
17 {% endblock %}
```

Figure 9.8: Templating using variables

Here, the line numbers 9 – 12 result in a clickable link to another page. Let's see how this happened. The href tag of HTML sets the value returned by the variable `{{student.get_absolute_url}}` as the link to the page and the tag of HTML sets the value returned by variables `{{student.first_name}}` and `{{student.last_name}}` as the display text of the link. So, HTML tags and Python tags work hand in hand.

Fetching values from different data types:

List: `{{my_list.index}}`

as shown in the example.

Tags:

Syntax: `{% tag_name %}`

Like any other markup language, the Django Templating Language has its set of tags. These tags are used to incorporate some logic in and execution to HTML templates with the help of the Python code in them.

Django has a big list of inbuilt tags and is documented in detail at [https://docs.djangoproject.com/en/3.2/ref/templates/builtins/](#). We will browse through some of the tags used in our projects:

extends: This tag is used to inherit parent templates as discussed in the section *Template inheritance* of this chapter.

block: This tag is used to enforce template inheritance. So, anything which has to be overridden by the child template is written in block tags. This tag needs to be closed. For example: In list.html of your project, the title and content were to be overridden by the child template and therefore, they were written inside the `{% block title %} {content} {% endblock %}` tags.

for: This tag is used to implement for loop to iterate over any iterable. This tag also must be ended by `{% endfor %}`.

Example: In list.html of your project, you iterated over a list of students.

empty: This is an optional tag that can be used along with the for tag. This tag is invoked if the iterable provided does not contain

anything (or length of the list is 0).

Example:

```
{% for item in object_list %}
    <li>
        <h2><a href="{{ student.get_absolute_url }}">
            {{ student.first_name }} {{ student.last_name }}
        </a>
    </h2>
    <p class="date"> Enrollment Number {{ student.student_id }} </p>
</li>
{% empty %}
    <li>No Students yet.</li>
{% endfor %}
```

Figure 9.9: Use of the empty tag

if, else, and, or, not: These tags can be used to implement if-else conditional operations. The example can also be implemented using if-else tags as shown in the following screenshot:

```
<ul>
    {% if students %}
        {% for student in students %}
            <li>{{ student.name }}</li>
        {% endfor %}
    {% else %}
        <li>Sorry, no students in this list.</li>
    {% endif %}
</ul>
```

Figure 9.10: Conditional statements in templates

Filters:

|date:"Y-m-d"}}

This tag is used to format the variables or tag arguments for displaying in a specific format on the HTML page.

default:

|default:"2020-01-01"}}

default_if_none:

```
| default_if_none:"Y-m-d"}}
```

time:

Take a look at the following table for reference:

reference: reference: reference:

reference:

reference:

reference: reference:

reference: reference: reference: reference: reference: reference: reference: reference:
--

Table 9.1: Characters and their descriptions for Django templates.

| time:"H:i:s"}} output will be like : 09:39:49

date:

Look at the following table:

table: table:

table: table: table:

table: table: table:

table:

table:

table: table: table: table: table:

table: table: table: table: table:

table:

table:
table:

table:

Table 9.2: Characters and their descriptions for Django templates

| time:"Y-m-d"}} output will be like : 2020-01-01

There are more filters you can check out in the official documentation mentioned earlier.

Comments:

This is comment #}

You can use this tag to write a multiline comment which won't be rendered to the HTML page.

Conclusion

So, now you have a better understanding of templates that we have been writing all this while. You now know how to write templates and would be able to write your own templates. For practice, just imagine any response context dict, etc.) and write templates based on that. Do refer to the Django's official documentation for templates and filters. You learned about models, views, and templates in detail to work on any major project. Of course, you will have to refer to the Django documentation for more advanced features.

In the next chapter, you will read about Forms in Django. You will learn how to create forms and obtain data from them.

Questions

What are templates?

Which templating languages are supported by Django?

Which tag is used to create content?

Explain with an example – how to write a for loop in the template?

Explain with an example – how to create a nested loop in a template?

What is template inheritance?

URLs and Regex

The bridge between a request from the browser and the view which will serve that request is the url-patterns. The patterns are written using *regular expressions*. You can use the name of these patterns to create hyperlinks in the webpages.

It is important that you read about writing url-patterns so that you can connect your request from the browser to the correct views. If the patterns are incorrect, you will get a lot of pages not found errors.

In this chapter, you will read about the regex also known as regular expressions. You will learn how to write and interpret regex so that you can easily write a url-pattern for any expected URL. You will read about other methods and functions available in the URLs module.

Structure

Introduction

Regex

Functions available in URLconfs

Conclusion

Objectives

To learn about the functions available for writing url-patterns.

Learning regex so that complex patterns can be written.

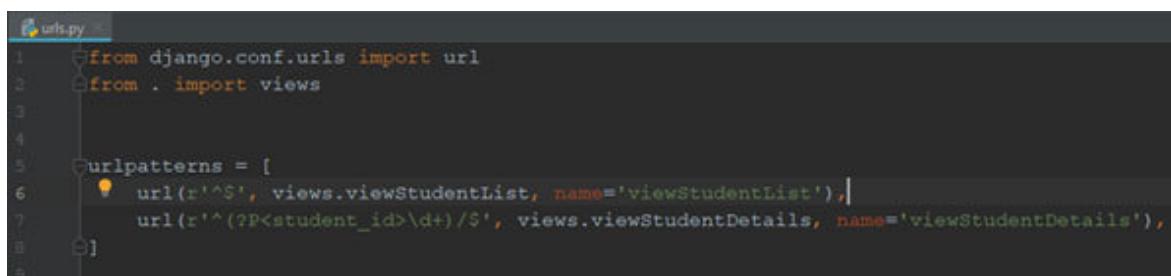
Practicing some url-patterns.

[*Introduction*](#)

Let's pull out the models.py file from your project from the previous chapter and see what have used already. Briefly, one can say that the models.py file may contain the import statements and model class. Further, the model class contains several fields, subclasses, and methods. This is a very basic structure of a models.py file. It can contain several model classes. Further, these classes can have several methods and subclasses. Let's take a look at it in more detail.

Functions available in URLconfs

You must have noticed that we use the url method to write our url patterns. The following screenshot displays a sample urls.py file with some url-patterns. Take a detailed look at the following screenshot:



```
urls.py
1 from django.conf.urls import url
2 from . import views
3
4 urlpatterns = [
5     url(r'^$', views.viewStudentList, name='viewStudentList'),
6     url(r'^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
7 ]
```

A screenshot of a code editor showing a Python file named 'urls.py'. The file contains Django URL configuration code. It imports 'url' and 'views' from 'django.conf.urls' and '.'. It defines a list 'urlpatterns' containing two 'url' statements. The first 'url' statement has a regex '^\$' and points to 'views.viewStudentList' with name 'viewStudentList'. The second 'url' statement has a regex '^(?P<student_id>\d+)/\$' and points to 'views.viewStudentDetails' with name 'viewStudentDetails'. The code is syntax-highlighted with colors for different parts like keywords, comments, and strings.

Figure 10.1: Use of the url method

There are other functions available which you can use for this purpose. In the latest versions of Django, the re_path function has been introduced which is similar to the url function. The url function will be removed from Django in future versions. So, it is important that you have the knowledge of all functions available for writing url-patterns. All these functions are available in the django.conf.urls module.

Note: You must know that the regex or the pattern is everything that follows the domain.

Example: For your localhost, the domain will be

If the URL is the urls.py will try to match “ ” (no characters) with any of the url-patterns. If the URL is the urls.py will try to match for ‘student/14’. For your site: the domain will be

If the URL is the urls.py will try to match “ ” with any of the url-patterns. If the URL is the urls.py will try to match for student/14 with any of the url-patterns.

`url()/re_path(): re_path(route, view, kwargs=None, name=None)`

You have been using the `url()` function to define the The arguments of this function are as follows:

`route`: It is a string that contains the regular expression or the regex which would be able to match with the expected URLs.

`view`: The view (class-based or function-based) which needs to be invoked.

`kwargs`: These are used when you have additional stuff to pass to the view function.

`name`: This is a name given to the url-pattern. This is helpful in URL reversing.

Examples: The urls.py in [Figure 10.1](#) uses the `url()` method.

```
path(): path(route, view, kwargs=None, name=None)
```

The arguments of this function are as follows:

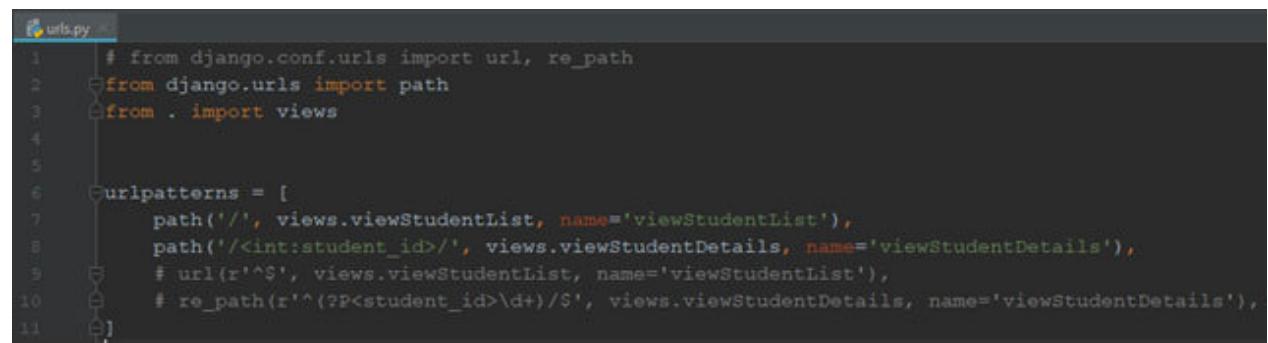
route: It is a string that contains the regular expression or the regex which would be able to match with the expected URLs.

view: The view (class-based or function-based) which needs to be invoked.

kwargs: These are used when you have additional stuff to pass to the view function.

name: This is a name given to the url-pattern. This is helpful in URL reversing.

Example:



```
urls.py
1  # from django.conf.urls import url, re_path
2  from django.urls import path
3  from . import views
4
5
6  urlpatterns = [
7      path('/', views.viewStudentList, name='viewStudentList'),
8      path('/<int:student_id>/', views.viewStudentDetails, name='viewStudentDetails'),
9      # url(r'^$', views.viewStudentList, name='viewStudentList'),
10     # re_path(r'^^(?P<student_id>\d+)/$', views.viewStudentDetails, name='viewStudentDetails'),
11 ]
```

Figure 10.2: Use of path method

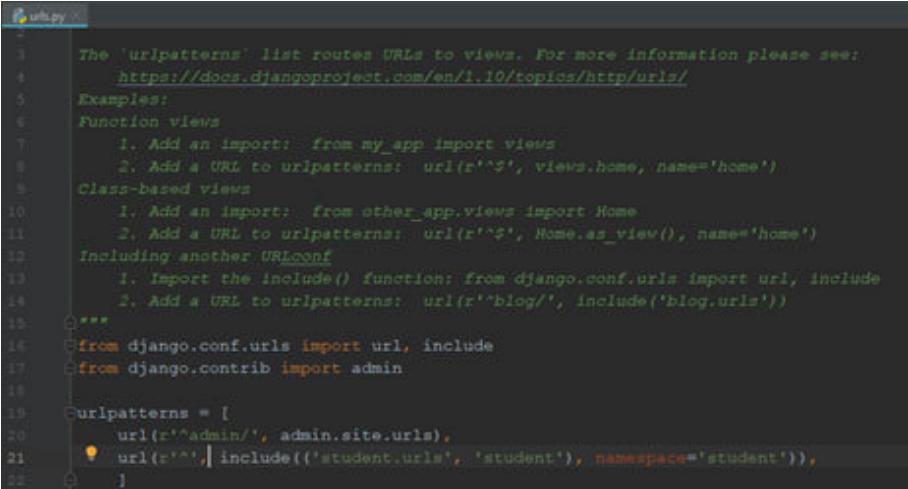
include()

Usage:

```
include(module, namespace=None)
include(pattern_list)
include ((pattern_list, app_namespace), namespace=None)
```

This function is used to include a set of url-patterns written or defined elsewhere into a url-pattern. You may have used this when you imported the url-patterns of your app student into the url-patterns of your project:

Example:



```
urls.py
1  The 'urlpatterns' list routes URLs to views. For more information please see:
2      https://docs.djangoproject.com/en/1.10/topics/http/urls/
3  Examples:
4      Function views
5          1. Add an import: from my_app import views
6              2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')
7      Class-based views
8          1. Add an import: from other_app.views import Home
9              2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')
10     Including another URLconf
11         1. Import the include() function: from django.conf.urls import url, include
12             2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))
13
14     from django.conf.urls import url, include
15     from django.contrib import admin
16
17     urlpatterns = [
18         url(r'^admin/', admin.site.urls),
19         url(r'^', include(('student.urls', 'student'), namespace='student')),
20     ]
```

Figure 10.3: Use of include function

These were functions commonly used while writing any Now, you must read about how to use regex to write the

Regex

A regular expression or regex is a string containing symbols and characters which are the representation of a certain string pattern. Regex is used to find or search for a string with a pattern.

Example: Mobile numbers should start with +91, have 10 characters, and all digits.

Email addresses should have

URL patterns should have

Such strings which have certain defined patterns can be represented by a regex and then the regex can be used to filter or find out if any given string follows the defined pattern. Let's see how you can write regex for any pattern. First, you need to know what special characters represent what type of string:

string:

string: string: string:

string: string:

string: string:

string: string: string:

string: string: string: string: string: string: string: string:

| string: string: string: |
| string: string: string: string: |
| string: string: string: string: string: string: string: string: |
| string: string: string: string: string: string: string: string: |
| string: string: string: string: string: string: string: string: |

string: string: string: string: string: string: string: string:

| string: string: string: string: string: string: string: string: |
| string: string: string: string: string: |
| string: string: string: string: string: string: string: string: string: |
| string: string: string: string: |
| string: string: string: string: |
| string: string: string: string: string: string: string: string: string: |
| string: string: string: string: string: string: string: |
| string: string: string: string: string: |

Table 10.1: Regex characters and descriptions

It is not important that you remember all these, but you must have these handy while writing URL patterns. This will help you a lot in creating a complex regex.

Writing a regex for different url-functions

You can read about a regex and different url-functions. Now, let's see how to write url-patterns for different url-functions.

url/re_path()

The url or the re_path takes a regex as a raw string. Python has two types of strings – string written inside ‘ ‘ and raw string written as r’ ‘. These functions take the raw string regex; thus, you will see the r in front of the regex. Let’s take an example:

```
urlpatterns = [
    url(r'^$', views.viewStudentList, name='list'),
    re_path(r'^(?P<student_id>\d+)/$', views.viewStudentDetail, name='detail')
]
```

Figure 10.4: Regex for url function

Here, in the second url-pattern:

r: denotes a raw string

: start of raw string

^: indicates the start of the url-pattern.

(?P\d+): This is one entity. Let’s see what does each one does individually:

This indicates that the following string inside angle brackets <> is the variable_name which will fetch a certain value from the next part of the URL.

: It is variable which will store the ID of the student.

\d: This is for any digit (0-9).

+: one or more occurrences of \d. Together with \d, it becomes \d+ which means any number of digits.

/: it means /

\$: This indicates the end of the url-pattern.

': This is the end of raw string.

This is how a regex is implemented. Now, try to write a url-pattern for the following URL:

<http://www.myinstitute.com/student/14/course/77>

Here, 14 is student_id and 77 is the course_id which needs to be filtered. So, let's start by the basic pattern which is: r'^/\$'. Now, adding the string after the domain makes it:

For the student_id: Again, the string makes it:

For course_id: r'^student/(?P\d+)/course/(?P\d+)/\$'. So finally, we have our regex for the given url-pattern is:

r'^student/(?P\d+)/course/(?P\d+)/\$'

path()

The new version of Django (2 and above) have the path function. This is easier than using the url function as we do have to write that complex regex. Let's see how this is handled:

```
urlpatterns = [
    path('', views.viewStudentList, name='viewStudentList'),
    path('<int:student_id>/', views.viewStudentDetail, name='viewStudentDetail'),
```

Figure 10.5: Regex for path function

Here, in the second url-pattern:

: This does the job of The data type int indicates that any integer present here would be passed to the variable

/: it means /

That's it, simple. This is how regex is implemented. Now, try to write a url-pattern for the following URL:

<http://www.myinstitute.com/student/14/course/77>

Here, 14 is student_id and 77 is the course_id which needs to be filtered. So, let's start by a basic pattern which is: '/'. Now, adding

the string after the domain makes it: ‘student/’.

For ‘student//’. Again, the string makes it:

For course_id: So finally, we have our regex for the given url-pattern:

‘student//course//’

You can read about both writing url-patterns using both the methods. Which one did you find easier? The path() method right. The patterns are pretty much straight forward and easy and these can be used in future. Practice more with

Conclusion

Now, you have a better understanding of what url-patterns are, how they work, the url-pattern matching, and the concept behind it. You learned how to write url-patterns using different methods like `url()` and These skills will enable you to map your requests to the correct views. You will be able to take advantage of the url-patterns to create hyperlinks in the upcoming chapters. You will get a better hang of it when you start coding.

Do browse through the official Django documentation about models to look for more specific code and latest updates. In the next chapter, you will read about forms. You will see how to take the input from users and save them in the database.

Questions

What are url-patterns?

What is a regex?

Which methods are available to write url-patterns?

How to add url-patterns from different apps into another urls.py file?

Write a url-pattern for the URL

www.basicssessions.com/23/rama/34/size/ using both path and url functions.

Forms in Django

If you are making a dynamic website, your site will need data from users. You can take inputs like username, password, content, and other relevant data about your website. Forms are used to fetch data from users.

Any dynamic application has to take a lot of data from users and this requirement makes it very important to read about forms. Django provides built-in forms. We just need to know how to import and use them.

In this chapter, you will read about forms in Django. To fetch any data from the user, we use forms. You have seen Django forms in action in the admin panel while entering student data. Of course, any application may have a requirement for taking input from users. Like signing in, queries, details, etc. Thus, it is important to learn about forms.

Structure

Introduction

Building basic forms

Form fields

Form validation

Model form

The form API

Conclusion

Objectives

Understanding how to create forms.

Understanding how to fetch data from forms.

Understanding how to create forms from the models in the project.

Introduction

HTML forms can also be used to take input from the user and can serve the same purpose. For this, we need to write the HTML code to create fields and capture input. You will have to write code for validating the data entered by the user, etc.

Django forms help in avoiding to write a lot of HTML code. Django forms can quickly generate the HTML form widgets. They can validate the data entered by the user and process it into a Python data structure. Most importantly, they can quickly create forms based on the models you have created. After all, you will be saving the data taken from the user into the models-database. So, you will need forms based on the fields in your models. Django forms make all this easy for you.

[Building basic forms](#)

To build Django forms, you need to create a forms.py file inside the app folder. In that file, you need to write the Python code to generate forms. It is very much like models. Then, you need to write a view for your form in the views.py file. You need to write a template for your form and link the template to the view. Also, you need to add the form url to url-patterns which will invoke the form view.

Let us do all of this with an example. We will create a new project for this chapter. This project will create a form where users can enter their name, email, and inquiries for a bookstore. Let us begin:

Create a new project and an app inside it. You can give any name to the project and app. I have named them basic_project and

Set up the project.

For templates, as you read in [Chapter 9: Django](#) we will follow the app_folder/templates/folder_with_appname structure. Inside the app folder, create a folder with the name templates. Inside the templates folder, create a folder with Inside the folder with create two HTML files – index.html,

Open the settings.py file of the project. Here, create a TEMPLATE_DIR for your templates and update the

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

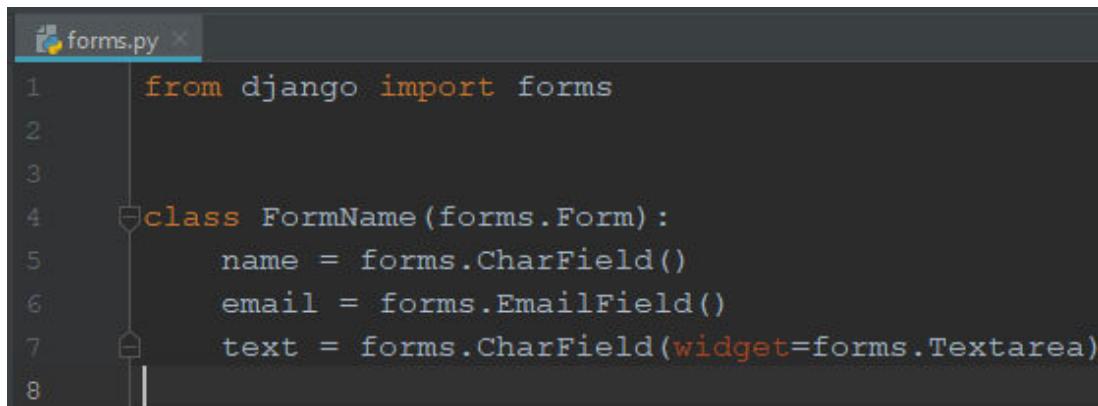
Figure 11.1: Template setting

In the settings.py file, add your app in the list of

Creating forms:

Create a Python file in the app folder with the name

Write the following code:



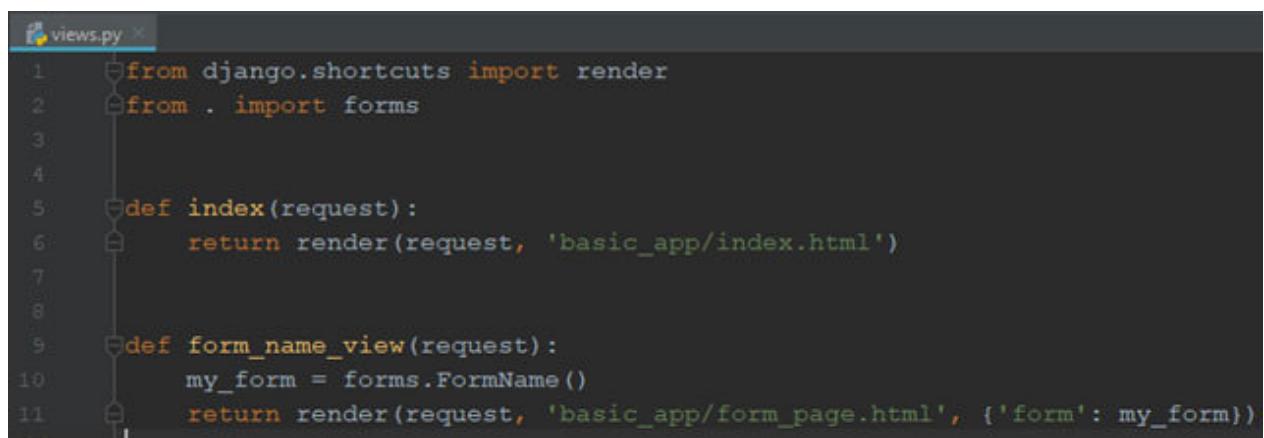
```
forms.py ×
1  from django import forms
2
3
4  class FormName(forms.Form):
5      name = forms.CharField()
6      email = forms.EmailField()
7      text = forms.CharField(widget=forms.Textarea)
8
```

Figure 11.2: Form class in forms.py file

Here, we have created a form class with the name This class inherits the forms.Form class. This is very much similar to models. Further, we have created three form fields – name, email, and text. Each of these fields has its field types similar to models. In the text field, you will see Widgets are used to create different types of input blocks on the frontend like checkboxes, etc.

Creating views:

We will create views for the homepage of this project and the form page. In the views.py file of the app, write the following code:



```
views.py
1  from django.shortcuts import render
2  from . import forms
3
4
5  def index(request):
6      return render(request, 'basic_app/index.html')
7
8
9  def form_name_view(request):
10     my_form = forms.FormName()
11     return render(request, 'basic_app/form_page.html', {'form': my_form})
```

Figure 11.3: View for the form in views.py

Let's understand the preceding code line by line.

In Line 2, we import the forms file so that we have access to the FormName class we created there.

In Lines 3 and 4, we can view the function for the homepage.

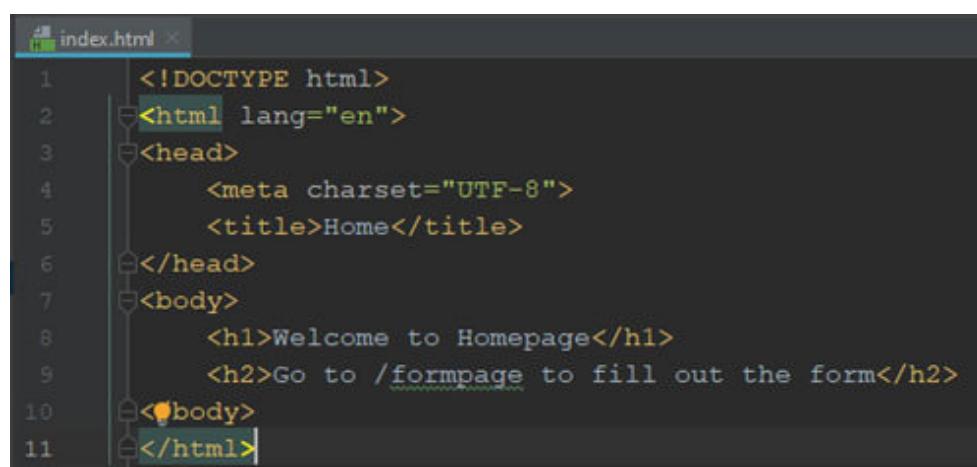
In Line 9, we create a view for the This view will be responsible to create a form and fetch data from users.

In Line 10, we create a form by executing forms.FormName() and store that form object in the variable This form object will be passed to the template to generate a form on the screen.

In Line 11, we pass the my_form to the template and render it as a response to the request as we have done earlier.

Creating templates:

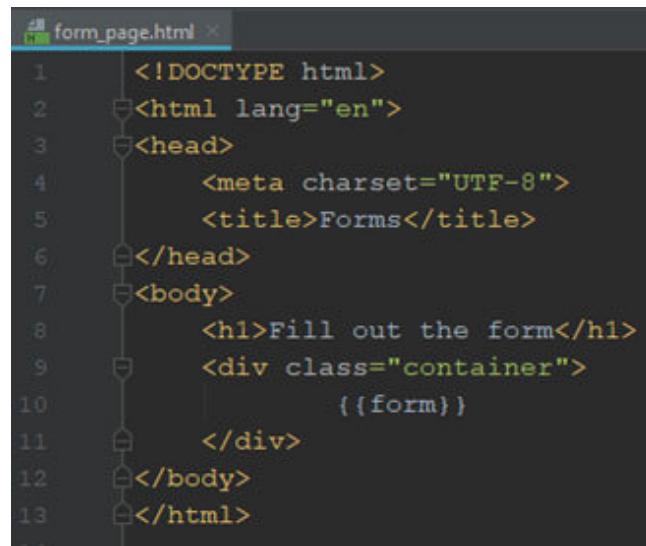
Open the index.html file and it will look like the following screenshot:



```
index.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Home</title>
6  </head>
7  <body>
8      <h1>Welcome to Homepage</h1>
9      <h2>Go to /formpage to fill out the form</h2>
10     <body>
11         </html>
```

Figure 11.4: Template

Open form_page.html and write the following code in the file:



```
form_page.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Forms</title>
6  </head>
7  <body>
8      <h1>Fill out the form</h1>
9      <div class="container">
10         {{form}}
11     </div>
12  </body>
13 </html>
```

Figure 11.5: Template

Both the templates are simple. In we simply take the my_form object passed to the template by the view and display it on the screen.

Updating the urls.py file:

Open the urls.py file of the project and it will look like the following screenshot:

```
from django.contrib import admin
from django.urls import path
from basic_app import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index'),
    path('index/', views.index, name='index'),
    path('formpage/', views.form_name_view, name='form_name'),
]
```

Figure 11.6: *Urls.py file*

Now, open the terminal and execute the makemigrations and migrate commands. Then, run the runserver command and open your localhost in the browser. You will see something like the following screenshot. It's not a much decorated form; although a form is generated without writing any significant HTML code. You can note down the input block for the text field. It is a widget that we defined in our form class in



Figure 11.7: *Application up and running*

It's a very basic form that does nothing as of now. In the next section, you will read about submitting and fetching the data entered by users in the form.

Fetching data entered in the forms

Now, we will enhance our forms project and see how we can obtain data from the forms and use it:

Update view:

Open the views.py file and update the form_name_view as shown in the following screenshot:

```
 9  def form_name_view(request):
10     if request.method == 'POST':
11         my_form_with_data = forms.FormName(request.POST)
12         if my_form_with_data.is_valid():
13             # Picking data entered by user.
14             print('Form Validated')
15             print("Name: " + my_form_with_data.cleaned_data['name'])
16             print("email: " + my_form_with_data.cleaned_data['email'])
17             print("text: " + my_form_with_data.cleaned_data['text'])
18         my_form = forms.FormName()
19     return render(request, 'basic_app/form_page.html', {'form': my_form})
20
```

Figure 11.8: Fetching data from forms

Let's understand the preceding code line by line.

Line 10: Here, we check whether the request made by the browser is POST request or not. There are two types of https requests: GET and To obtain data from the user, we need the POST request. The POST request indicates that the request will have data to fetch.

Line 11: Here, we create an object of the form class and use that object to fetch data from the request and save it to the variable

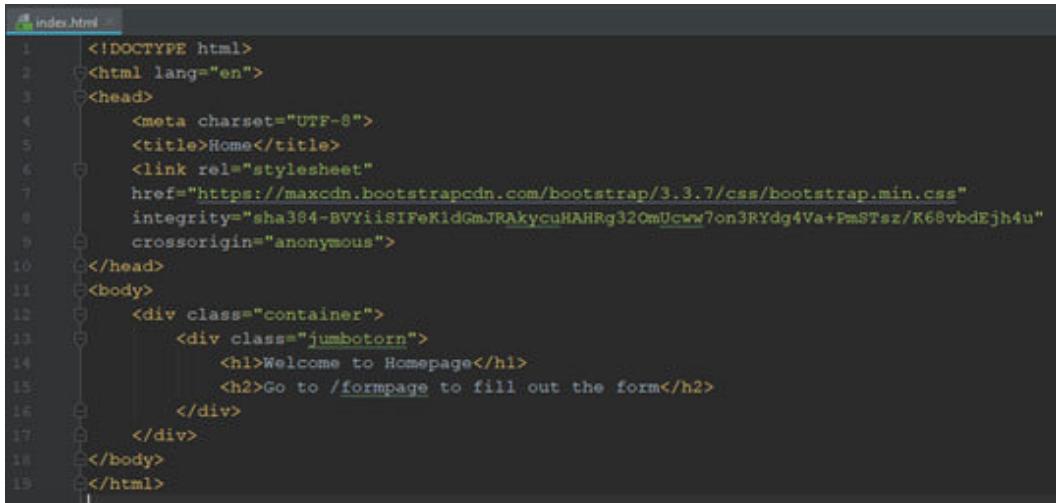
Line 12: Here, we check whether the data entered by the user is as per our field type. For example, the email field should have data in the email field format, etc.

Lines 14 – 17: Here, we obtain data from the request using the variable my_form_with_data. Now, here we are simply printing the data. Once we have the data, it can be used as per our requirement like saving it to a database, making calculations and returning any result, etc.

Lines 18 – 19: These lines are to be executed as mentioned earlier. The my_form variable will have no data and will be rendered to the screen. Thus, the fields will be blank once the user submits the form. If you wish the data to be present in fields even after the user submits the form, move the code inline 18 to just above line 10 and rename the my_form_wth_data to my_form variable.

Creating templates:

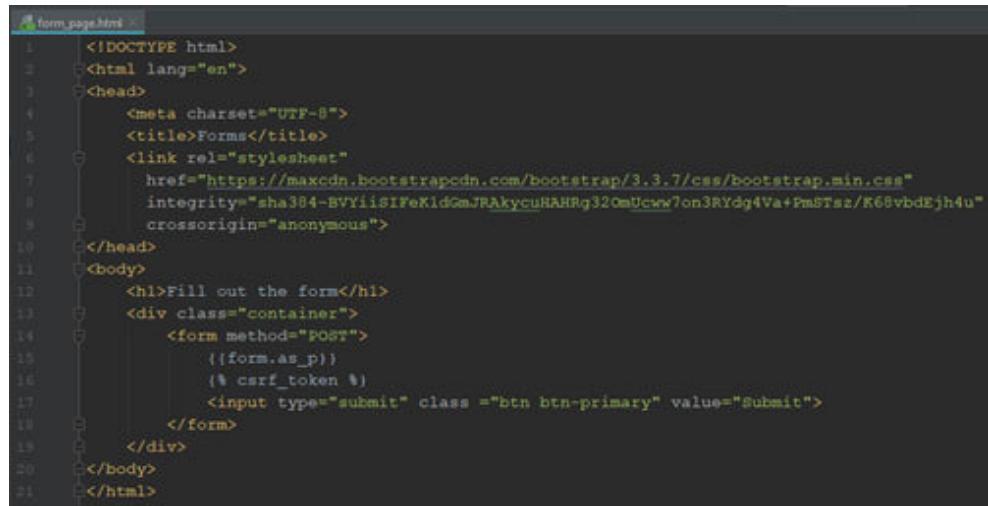
Let's beautify our webpage and add some styles. Open the index.html file and the file should like the following screenshot:



```
index.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Home</title>
6      <link rel="stylesheet"
7          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
8          integrity="sha384-BVYiiSIFeKIdGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
9          crossorigin="anonymous">
10 </head>
11 <body>
12     <div class="container">
13         <div class="jumbotron">
14             <h1>Welcome to Homepage</h1>
15             <h2>Go to /formpage to fill out the form</h2>
16         </div>
17     </div>
18 </body>
19 </html>
```

Figure 11.9: Template

Open form_page.html and write the following code in the file:



```
form_page.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Forms</title>
6      <link rel="stylesheet"
7          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
8          integrity="sha384-BVYiiSIFeKIdGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
9          crossorigin="anonymous">
10 </head>
11 <body>
12     <h1>Fill out the form</h1>
13     <div class="container">
14         <form method="POST">
15             {{form.as_p}}
16             {% csrf_token %}
17             <input type="submit" class="btn btn-primary" value="Submit">
18         </form>
19     </div>
20 </body>
21 </html>
```

Figure 11.10: Template for form

Here, apart from beautifying the template, we made the following changes:

Line 15: We have replaced the form by You must have noticed that the form fields were displayed from left to right, next to each other. After replacing, the fields will be displayed from top to down – like a paragraph.

Line Reference Token (CSRF) is a security token needed to ensure that the data you have entered is captured by the original website and not captured by any other site. Without this token, Django will not be able to submit the form and will throw an error.

Line 14: Here, we indicate that the request will be a POST request. Thus, when the submit button is pressed, a POST request with data entered by the user is triggered.

Line 17: Here, we add a **Submit** button to the form.

Now, execute the runserver command and check how your form looks like:

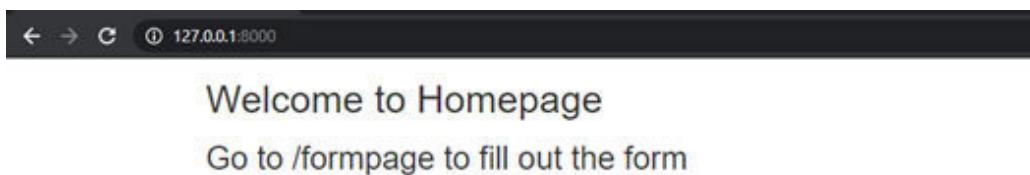


Figure 11.11: Homepage of application

Change the URL to 127.0.0.1:8000/formpage/ and you will see the form as shown in the following screenshot:

The screenshot shows a web browser window with the title 'Forms' and the URL '127.0.0.1:8000/formpage/'. The main content is a form titled 'Fill out the form'. It contains three input fields: 'Name' with the value 'Awanish', 'Email' with the value 'awa@gmail.com', and a larger 'Text' area containing the message: 'Hi
I need information about how to buy Django book.
Thank You
Awanish'. Below the text area is a 'Text:' label. At the bottom is a blue 'Submit' button.

Figure 11.12: Form page of the application

Go back to the terminal and you will see the data you have filled in the form. The following screenshot displays a sample of the data entered:

```
System check identified no issues (0 silenced).
May 04, 2020 - 17:55:10
Django version 2.2.2, using settings 'basic_forms.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[04/May/2020 17:55:25] "GET / HTTP/1.1" 200 532
[04/May/2020 17:56:11] "GET /formpage/ HTTP/1.1" 200 1006
Form Validated
Name: Awanish
email: awa@gmail.com
text: Hi
I need information about how to buy Django book.
Thank You
Awanish
[04/May/2020 17:59:21] "POST /formpage/ HTTP/1.1" 200 1117
```

Figure 11.13: Data fetched from the form printed on the console

In the console, you can see the data entered in the form that has been fetched and printed by

Form fields and arguments

Just like the models, forms also have fields' types and those fields' types have arguments. These can be used to fetch data from the user in various ways.

Form field arguments

Here is a list of arguments which can be used to modify the form fields:

required: This argument is used to set if the field is required or not. It is set as True by default. You can set it to false for optional fields.

Example:

```
name = forms.CharField(required=False)
```

label: On the webpage, the form field name is picked by the field name defined in the forms class in the forms.py file. If you want any other name, define it as follows:

```
name = forms.CharField(label='Enter your name')
```

label_suffix: This argument is used to set a label suffix like ‘-’, ‘=’. As of now in your project, the default label is label_suffix is ‘:’.

Example:

```
name = forms.CharField(label_suffix=' =')
```

initial: If you want any field to have a pre-filled value, the user can choose to enter data along with the pre-filled value or erase the pre-filled value. You can use this argument for this purpose.

Example:

```
name = forms.CharField(initial='Mr. ')
```

widget: As mentioned earlier, widgets are used to have a certain type of input block. A detailed documentation is present at [Take a look at the type of widgets that are available.](#)

This is the text shown to the user who has entered incorrect data and submitted the form.

Example:

```
text = forms.CharField(max_length=100, help_text='100 characters max.')
```

error_messages: When any field with incorrect data is entered, then an error is thrown after validation. You can edit the `error_message` here.

Validators: This is a list of validators that will be applied when Django validates the form. For example, `EmailValidator`, etc. You can find a detailed documentation of these validators at [here](#).

Localize: It is True by default. If all the stuff which falls under internationalization will be according to the local.

Disabled – True or False: If True, the field will be disabled and the user won't be able to edit. The field is greyed out.

Form fields

You read about the forms of field arguments. Let's check out the field types available for forms in Django. The form field types available in Django are shown in the following table:

table:

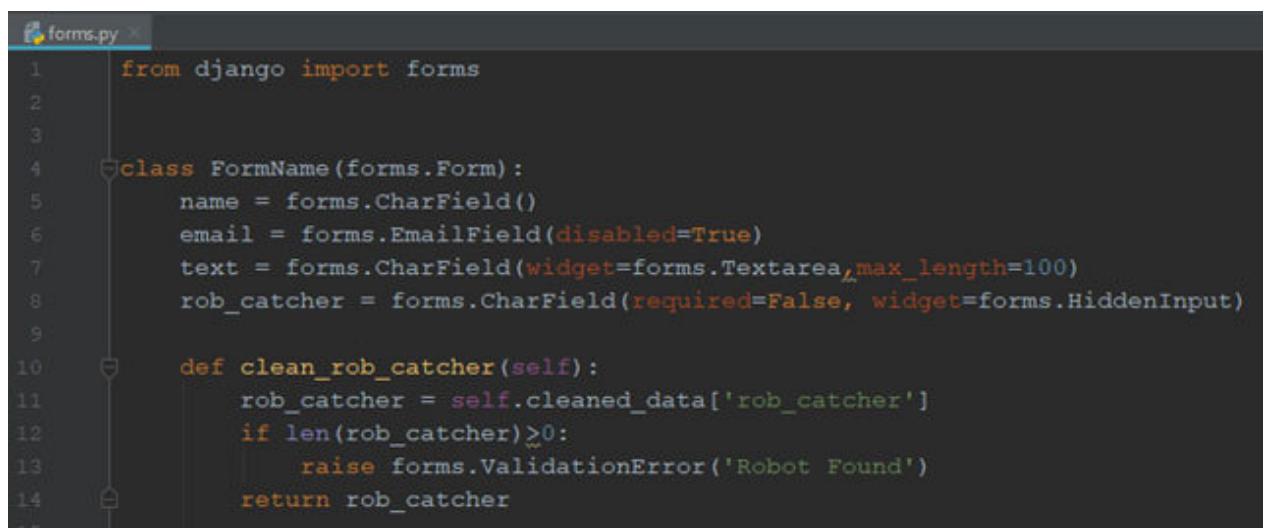
You have already used some of these field types in your project. The usages of other field types are very similar to these. A detailed documentation about the built-in field types can be found at [You can refer to this documentation.](#) Look for the default

values of different arguments of these field types to be sure of what will be the outcome of using any field type.

Form validation

Validation means to validate the data entered by the user. Generally, in advance projects, the frontend is advanced and it takes care of the data entered in the forms. It can also be done from the backend. Django offers a whole set of in-built Form Validators and the documentation can be found at [In this section, you will read about how to use some of these validators.](#)

Let's take an example of form validation. Take a look at your form.py file and it should look like the following screenshot:



```
forms.py
1  from django import forms
2
3
4  class FormName(forms.Form):
5      name = forms.CharField()
6      email = forms.EmailField(disabled=True)
7      text = forms.CharField(widget=forms.Textarea, max_length=100)
8      rob_catcher = forms.CharField(required=False, widget=forms.HiddenInput())
9
10     def clean_rob_catcher(self):
11         rob_catcher = self.cleaned_data['rob_catcher']
12         if len(rob_catcher)>0:
13             raise forms.ValidationError('Robot Found')
14         return rob_catcher
15
```

Figure 11.14: Writing validation for forms

Here, we are adding a field named This field will be used to check whether any robot is trying to access your page. Set the required field option to False – this will make it a non-mandatory field. Set the widget to HiddenInput – this will hide the field on the page

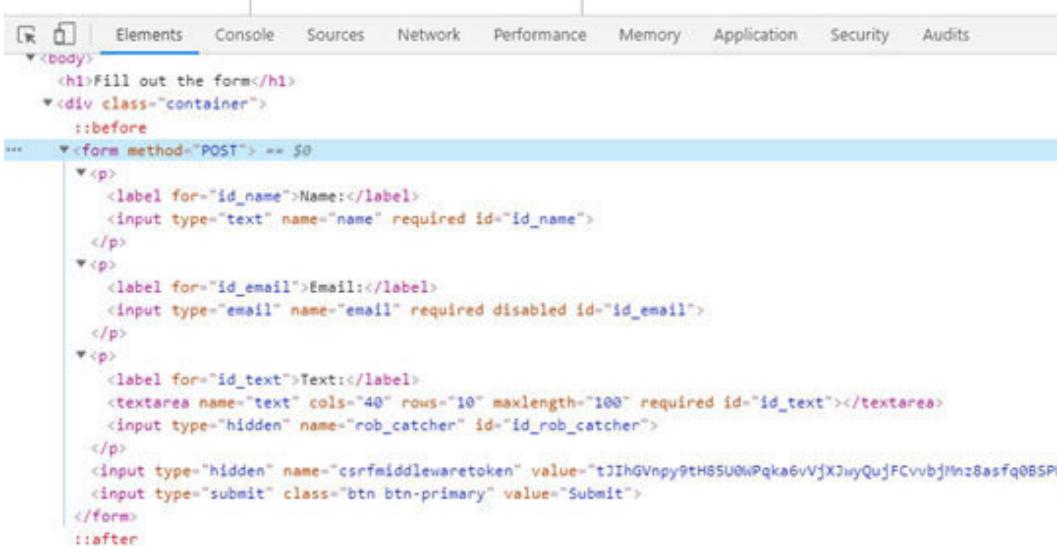
and thus, the human users won't be able to see it; this won't fill any data in this field. On the other hand, robots crawl through the HTML of the page and look for input fields and fill them. In this case, a robot will get an input field `rob_catcher` and fill some data in it.

Thus, to differentiate between human users and robots, all we need to do is to check whether the `rob_catcher` field has received any input or not. We need to perform this check before consuming the data in the other fields and stop further processing as the request was not made by a genuine human user.

Now, let's take a look at the `clean_rob_catcher` function. This function is a validator. When Django finds such a method in `forms-class`, it looks for a field which matches with the name after `clean_` and uses the function to validate the field matched. So, here `clean_rob_catcher` matches to `Now`, this field will be validated. We know that this field should not have any data.

So first, we will use the `cleaned_data` method to fetch the data entered into the field. Then, check whether its length is more than zero. If yes, some data has been filled and the robot has used our page. In this case, we need to raise a validation error. Let's test this.

Once you have updated the `forms.py` file, go to <http://127.0.0.1:8000/formpage/> and open Developers tools (On Chrome – press `F12` key), and you will see the HTML of your page. It will look like the following screenshot:



The screenshot shows the Chrome DevTools Elements tab. The DOM tree is displayed under the <body> node. A blue highlight is applied to the entire <form> element. Inside the <form> element, there are three <p> elements. The first <p> contains a label for "id_name" and an input field with type="text" and id="id_name". The second <p> contains a label for "id_email" and an input field with type="email" and id="id_email". The third <p> contains a label for "id_text" and a text area with name="text". Below these is a hidden input field with type="hidden" and name="rob_catcher" and id="id_rob_catcher". At the bottom of the <form> is a submit button with type="submit" and class="btn btn-primary" and value="Submit". The entire <form> element is enclosed in a <div> with class="container". There are also <:before> and <:after> pseudo-elements at the top and bottom of the <div> respectively.

```
<h1>Fill out the form</h1>
<div class="container">
  <:before>
  ... <form method="POST"> == $0
    <p>
      <label for="id_name">Name:</label>
      <input type="text" name="name" required id="id_name">
    </p>
    <p>
      <label for="id_email">Email:</label>
      <input type="email" name="email" required disabled id="id_email">
    </p>
    <p>
      <label for="id_text">Text:</label>
      <textarea name="text" cols="40" rows="10" maxlength="100" required id="id_text"></textarea>
      <input type="hidden" name="rob_catcher" id="id_rob_catcher">
    </p>
    <input type="hidden" name="csrfmiddlewaretoken" value="tJlhGVnpy9tH85U0wPqka6vVjXJiyQujFCvubjMnz8asfq0BSPi"
    <input type="submit" class="btn btn-primary" value="Submit">
  </form>
  <:after>
```

Figure 11.15: HTML of the form page

Here, you can see the input fields – Name, Email, and the field with id = This is the field you will not be able to see on the web page. Now, let's try to fill in this hidden field like a robot. Edit the HTML and add a value tag as shown in the following screenshot:



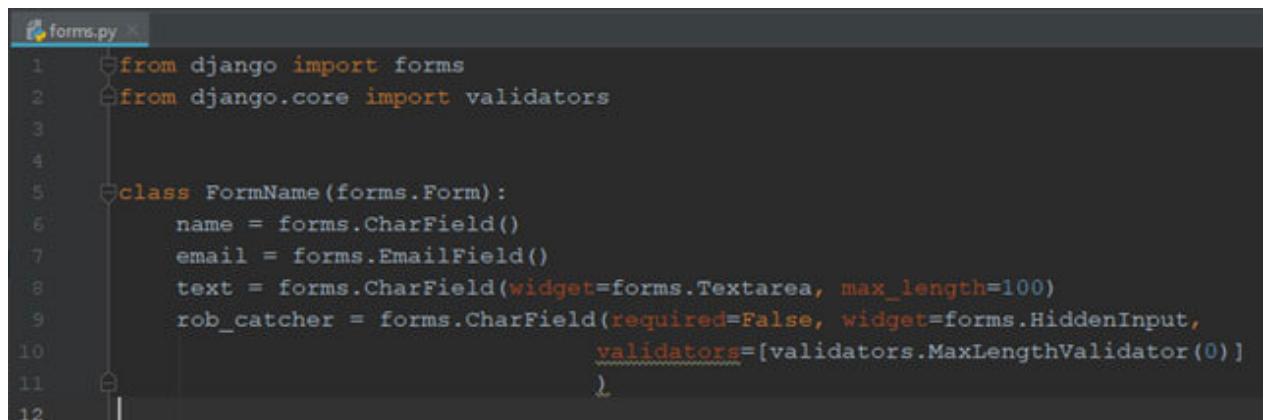
The screenshot shows the same DOM structure as Figure 11.15, but with a change to the hidden input field. The <input type="hidden" name="rob_catcher" id="id_rob_catcher"> line now includes a <value="Hello"> tag, which is highlighted with a blue selection bar. The rest of the code remains the same.

```
<label for="id_text">Text:</label>
<textarea name="text" cols="40" rows="10" maxlength="100" required id="id_text"></textarea>
<input type="hidden" name="rob_catcher" id="id_rob_catcher" value="Hello"> == $0
</p>
```

Figure 11.16: Adding value to the hidden field

Now, fill the other details and click on You will notice that this time nothing gets printed on the console. This is because the Validation error was invoked. Now, you know how write your validations. You would be able to write your validations for each of your fields.

Now, let's check out the in-built validators. We will implement the same scenario using a built-in validator. Update your forms.py file so that it looks like the following screenshot.liket

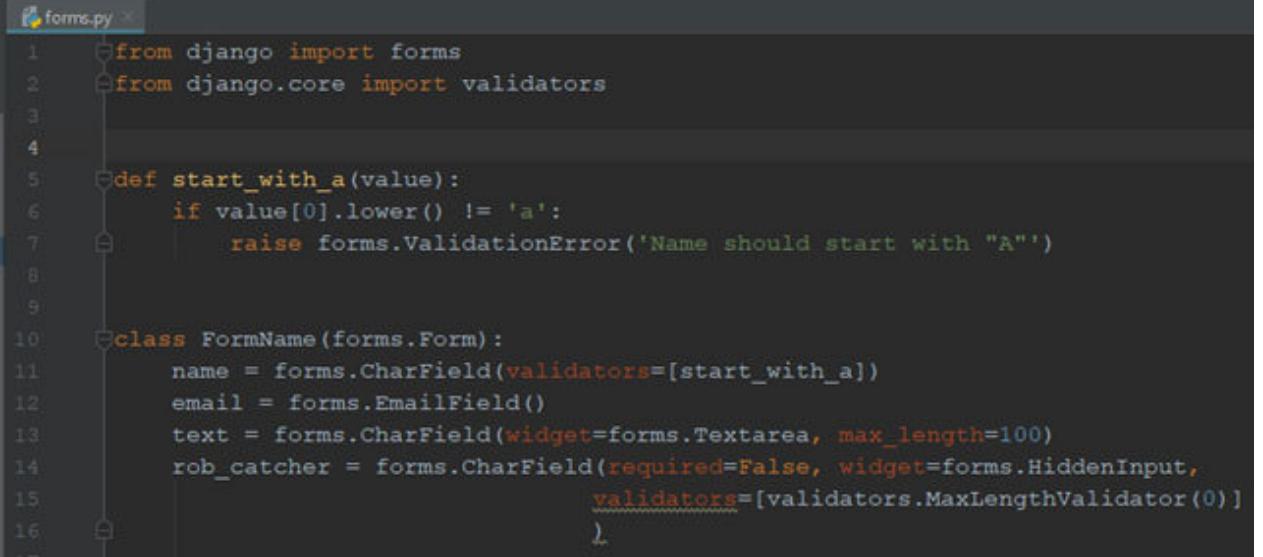


```
1 from django import forms
2 from django.core import validators
3
4
5 class FormName(forms.Form):
6     name = forms.CharField()
7     email = forms.EmailField()
8     text = forms.CharField(widget=forms.Textarea, max_length=100)
9     rob_catcher = forms.CharField(required=False, widget=forms.HiddenInput,
10                                    validators=[validators.MaxLengthValidator(0)])
11
12
```

Figure 11.17: Using built-in validation

Here, we imported the validators and used an in-built validator to the task of the function This is how you use built-in validators. Browse through the documentation and check out other validators.

Now, let's take a look at one more example of a different way of using validators. Create a validator which checks whether the name starts with A. Update your forms.py file so that it looks like the following screenshot:



```
forms.py
1  from django import forms
2  from django.core import validators
3
4
5  def start_with_a(value):
6      if value[0].lower() != 'a':
7          raise forms.ValidationError('Name should start with "A"')
8
9
10 class FormName(forms.Form):
11     name = forms.CharField(validators=[start_with_a])
12     email = forms.EmailField()
13     text = forms.CharField(widget=forms.Textarea, max_length=100)
14     rob_catcher = forms.CharField(required=False, widget=forms.HiddenInput,
15                                    validators=[validators.MaxLengthValidator(0)])
16
```

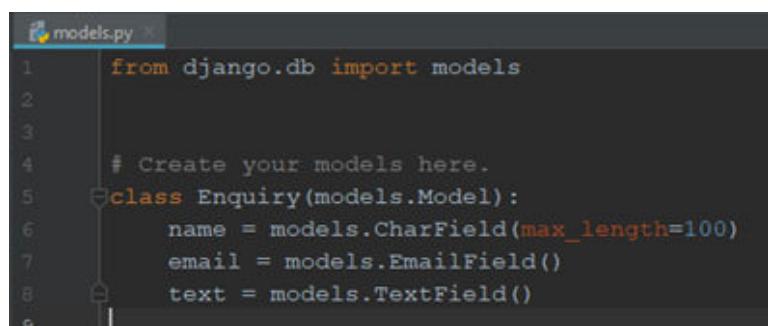
Figure 11.18: Using custom validators as the built-in validator

Here, we created a function that takes an argument and checks for a condition. The argument passed a value that is the HTML tag – value which contains the data filled in any field. Once you have such a method, you can use the validator field option and call your function as a validator of any number of fields.

Model forms

You can create forms and add validations to it. The real use of any form is when it can take data from users and store it into the database for further use. As you know Django interacts with the database using models. So, it becomes very important that we connect models with forms.

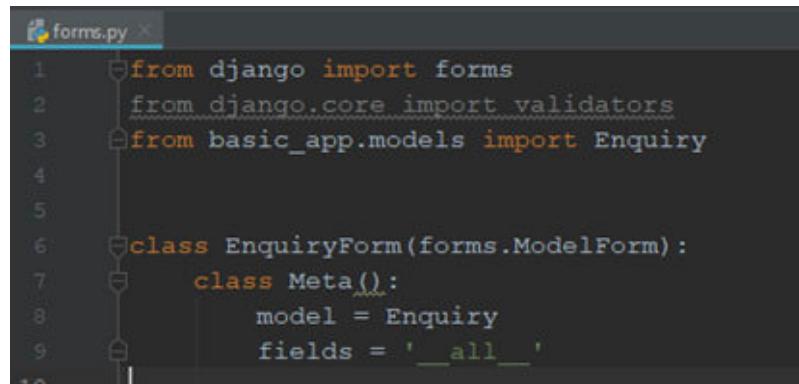
In most applications, you will have models and you will need to create a form that can take input from the user based on the fields of the model and save it into database. Like in this case, we are taking queries from users, but these queries must be saved somewhere so that they can be accessed and answered. So, let's make a model for it and connect it with our form. In the models.py file, type the following code:



```
models.py
1  from django.db import models
2
3
4  # Create your models here.
5  class Enquiry(models.Model):
6      name = models.CharField(max_length=100)
7      email = models.EmailField()
8      text = models.TextField()
```

Figure 11.19: Model for the form

Now, register this model in the admin.py file of your project. Refer to the previous chapters if you have any difficulty with registering. Now, you need to update your forms.py file to make it look like the following screenshot:



```
forms.py
1  from django import forms
2  from django.core import validators
3  from basic_app.models import Enquiry
4
5
6  class EnquiryForm(forms.ModelForm):
7      class Meta():
8          model = Enquiry
9          fields = '__all__'
```

Figure 11.20: Form for the model

Here, we imported the model for which the form is to be created. Then, we created a form. Notice that the form class inherits forms.ModelForm instead of forms.Form. Further in the Meta class, we define a variable ‘model’ and set it to the model name in the variable fields set This indicates that all the model class fields will have forms class fields. You can be selective here and create forms for selected fields as per your needs.

In remove the old form_name_view and write a new view as shown in the following screenshot:

```
views.py
1  from django.shortcuts import render
2  from basic_app.forms import EnquiryForm
3
4
5  def index(request):
6      return render(request, 'basic_app/index.html')
7
8
9  def enquiry_form_view(request):
10     form_empty = EnquiryForm()
11     if request.method == 'POST':
12         form_filled = EnquiryForm(request.POST)
13         if form_filled.is_valid():
14             form_filled.save(commit=True)
15             print('Form Validated')
16             print("Name: " + form_filled.cleaned_data['name'])
17             print("email: " + form_filled.cleaned_data['email'])
18             print("text: " + form_filled.cleaned_data['text'])
19             return render(request, 'basic_app/form_page.html', {'form': form_empty})
20         else:
21             print('Form Invalid')
22     return render(request, 'basic_app/form_page.html', {'form': form_empty})
```

Figure 11.21: View for connecting model and form

Here, the code is pretty much the same. The difference is line number 14. This command makes this view powerful. We are not only printing the information fetched from users but also saving it in the database. Corresponding to the view, also update the urls.py file:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index'),
    path('index/', views.index, name='index'),
    path('formpage/', views.enquiry_form_view, name='form_name'),
```

Figure 11.22: urls.py file for the form

Now, execute the following commands one by one and then test your application:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

```
python manage.py runserver
```

You have now created a small application for inquiry forms. This application is currently taking inquiries and saving it on the database. Here are a few tasks for you:

Add another field to a model named answered. This field should be a Boolean field. If the inquiry has been answered. Note that you can't use field = anymore in the Meta class.

Log in and configure your admin panel.

Create a page where all the inquiries can be seen along with their state (answered/unanswered).

Create a page where all the unanswered inquiries can be seen.

You should be able to perform these tasks. If not, please read the relevant sections again.

Conclusion

It has been a long chapter. You learned about forms, its fields, and form-field options. It is not expected of you to remember all these. You will get a hang of it when you start coding. For now, you must understand the concept of forms and how to link forms to models in Django. For a few topics, links to the official Django documentation is provided. You must be able to figure out the type of field, field option, and meta options. Relationships you must choose, or look for while creating forms.

Django offers so much that it is impossible to remember all of it. Thus, as a developer, you should always keep the Django documentation handy while coding. Go through the documentation and familiarize yourself with it. In the next chapter, you will read about user authentication and authorization.

Questions

What are the advantages of using Django forms?

Which template tag is used for displaying a form on the webpage?

How to create a built-in form?

What is form validation?

How to create a form using a model?

Setting Up Project

In the previous chapters, you read about different concepts of Django. You now understand the flow of a Django application, models, forms, templates, URLs, etc. You have been through the implementation of the individual components. If you have any confusion, read that chapter again. It is important that you are confident about the theoretical aspects of these components of Django.

In this chapter, we will begin with building a website from scratch and deploy it on the web for the world to see it. We will be using concepts from the chapters you have already read. By the end of this project, you will learn how to create websites using Django. If you have not gone through the previous chapters thoroughly, I would suggest reading them again. Moving forward from here onwards without the understanding of previous chapters may limit you to this project only and you may find yourself clueless when you start to work on other projects.

Structure

Introduction

Setting up the project

Updating the settings.py file

Conclusion

Objectives

To create a new Django project.

To update the settings.py file.

To create a superuser.

Introduction

We will create a website where users can log in and associate themselves to a different genre and post content and read posts of other users. Let's figure out the requirements for the project.

You need to think of two components: templates and databases. So, let's list down the webpages you want on the screen. It will have a homepage, login page, registration page, a page post list page, etc. Similarly, you will need models to store details of users, posts, etc. It seems very clumsy right! To keep it sorted, we segregate each of the functionalities into separate apps. One app for handling users and authentication, one app for posts, one app for genres, and so on. So, the website we are creating should have the following features:

User management feature: An app that should handle user registration, login, and logout functionalities. This feature should be responsible to ensure that any unauthorized user does not have access to view or modify posts.

Segregating content based on genre: An app that should help in categorizing the content and posts. If a user is associated with the genre, only then the content of that genre could be accessed.

Content or posts: An app to handle posts. This app should be able to add, remove, edit, delete, and render/display posts.

These functionalities are simple and can be developed easily. Since it's your first Django website, focus more on the concept here. Once you are done with this project, you should pick up more complex projects.

Setting up the project

We will begin with creating a root directory for your project which will contain everything. I am naming it Inside this folder, we will create a virtualenv for your project and activate it. Now, install Django. Django version 3.0.6 was installed here. We will create the Django-project inside the root folder.

You know the steps. Try to do it by yourself. Look at the following screenshots if you need help:

```
C:\Work\basicsessions-root>virtualenv bsenv
Using base prefix 'c:\\users\\awani\\appdata\\local\\programs\\python\\python37-32'
New python executable in C:\Work\basicsessions-root\bsenv\Scripts\python.exe
Installing setuptools, pip, wheel...
done.

C:\Work\basicsessions-root>bsenv\Scripts\activate.bat

(bsenv) C:\Work\basicsessions-root>pip install django
Collecting django
  Using cached Django-3.0.6-py3-none-any.whl (7.5 MB)
Collecting asgiref~=3.2
  Using cached asgiref-3.2.7-py2.py3-none-any.whl (19 kB)
Collecting pytz
  Using cached pytz-2020.1-py2.py3-none-any.whl (510 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.3.1-py2.py3-none-any.whl (40 kB)
Installing collected packages: asgiref, pytz, sqlparse, django
Successfully installed asgiref-3.2.7 django-3.0.6 pytz-2020.1 sqlparse-0.3.1

(bsenv) C:\Work\basicsessions-root>django-admin startproject basicproject
(bsenv) C:\Work\basicsessions-root>
```

Figure 12.1: Creating a virtual environment, installing Django and starting a project

Now, let us create the apps we discussed earlier. The design should be such that each app is independent and not dependent on other apps or projects. The code and configuration should be such that if you want to reuse any app again in any other project, you can simply copy the app folder and the job is done:

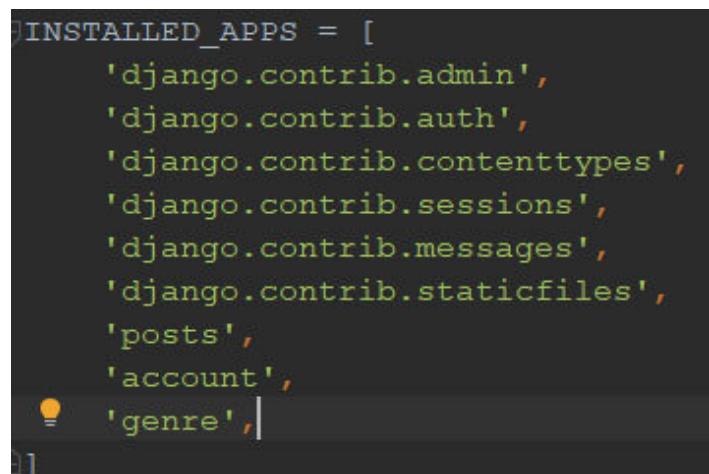
```
(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py startapp account  
(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py startapp genre  
(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py startapp posts  
(bsenv) C:\Work\basicsessions-root\basicproject>
```

Figure 12.2: Creating the account, genre and posts app

You can create these apps either now or later when you start coding for the app. Let us look at the project structure. The [Figure 12.5](#) shows the contents of different folders. The content in the app folders and are the same as of now. We will write different code into these folders. You can recall the usage of each of the files from previous chapters. We will write a similar code.

Updating the settings.py options

In [Chapter 3: Understanding Django](#) you read about the settings.py file and its configurations. Later, in different sections of the book, you read about different components of Django and how to configure them in the settings.py file:



```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'posts',
    'account',
    'genre',
]
```

Figure 12.3: Adding the app to installed apps in settings.py file

In our project here, we have created three apps. We must add them to the installed apps list. Let us do that.

Now, we will have templates for these apps. Remember that you read about how to configure the template directory in [Chapter 9: Django](#). Create a template directory and mention the path in the template configuration dict. Create the folder structure as app_name/templates/app_name for these apps. The structure shown in [Figure 12.5](#) for the account app must be replicated for the post

and genre app as well. Also, you will need a template folder for the template of your project.

You also read about static files in [Chapter 3: Understanding Django](#). This is the location where you store all the static content (.js, etc.) of your website. Just like templates, create a location for static files and save it a variable with the name

These are a few basic settings you would need to do in every project. Again, you should be to do all this on your own or refer to the following screenshot if you have doubts:

```
STATIC_URL = '/static/'  
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]  
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')  
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [TEMPLATE_DIR],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
            ],  
        },  
    },  
]
```

Figure 12.4: Adding template settings to settings.py file

This is all for the settings.py file for now. We will get back to it later if we need to make any configuration for any specific app.

Now, create a folder with the name static in the basicproject folder. Inside the static folder, create two folders css and js. Inside this css folder, create a master.css file. Now, inside the basicproject folder, create two folders css and js. These two folders will hold the styling and JavaScript files for the project. We have created quite a few files and folders. If you are confused, take a look at the following screenshot:

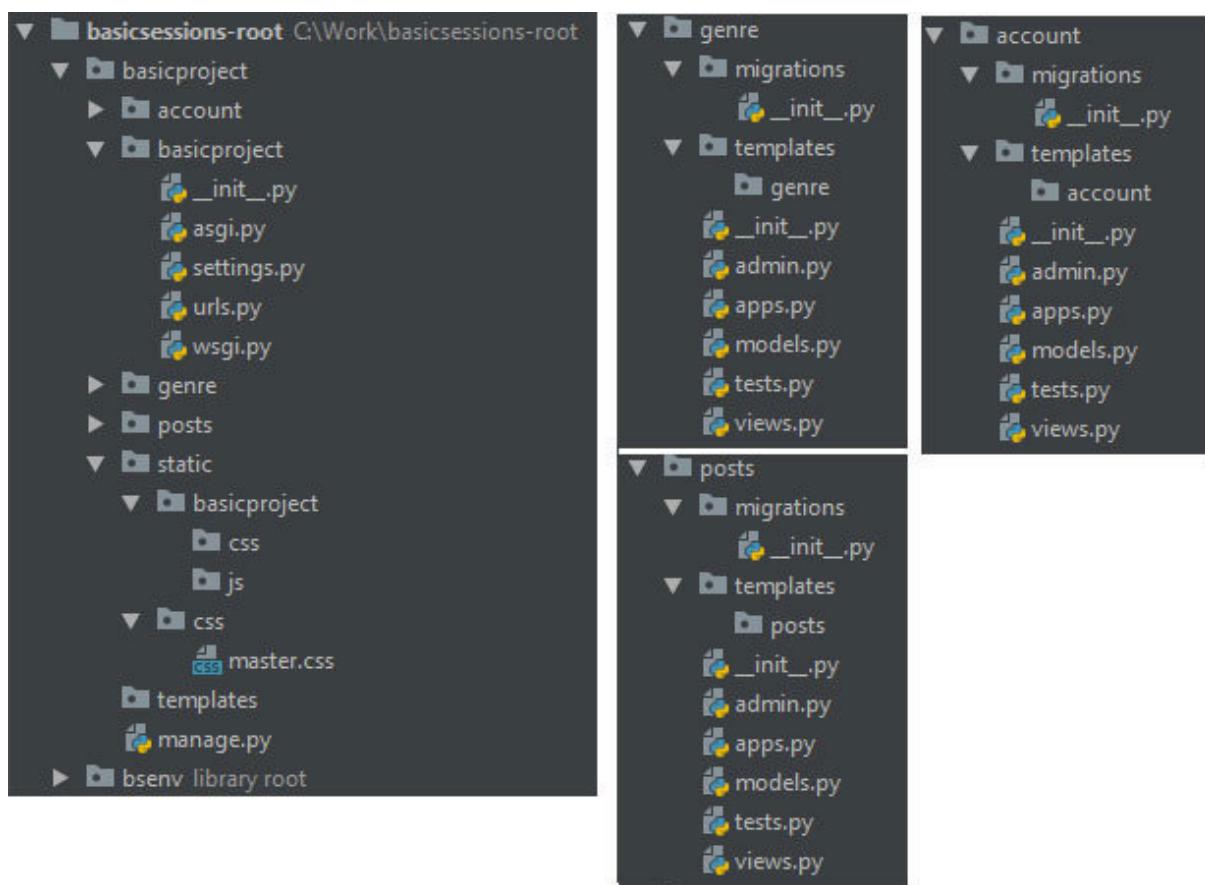


Figure 12.5: The file structure of the project and the apps inside it

The preceding screenshot contains the folder structure of the overall project and the individual apps. Compare your setup to the preceding structure and make sure you have this ready. You may

choose to have different names for your apps or project, but you must keep it relative.

Testing

Now that we have the setup, let's test what we have done so far. Execute the makemigrations command. You will get the message No changes detected:

```
(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py makemigrations
No changes detected

(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py makemigrations account
No changes detected in app 'account'

(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py makemigrations posts
No changes detected in app 'posts'

(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py makemigrations genre
No changes detected in app 'genre'
```

Figure 12.6: Making migrations

Now, run the migrate command. The migrations will be applied and a database will be created:

```
(bsenv) C:\Work\basicsessions-root\basicproject>python 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
```

Figure 12.7: Migrating changes

The database file will be named You can check this in your \\basicsessions-root\\basicproject folder or as shown in the following screenshot:

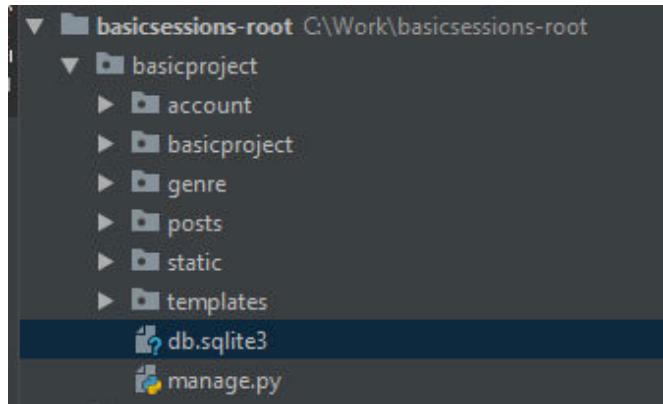


Figure 12.8: The db.sqlite3 file created after running migrations

Now, run the server. Open the URL mentioned on your console in your browser:

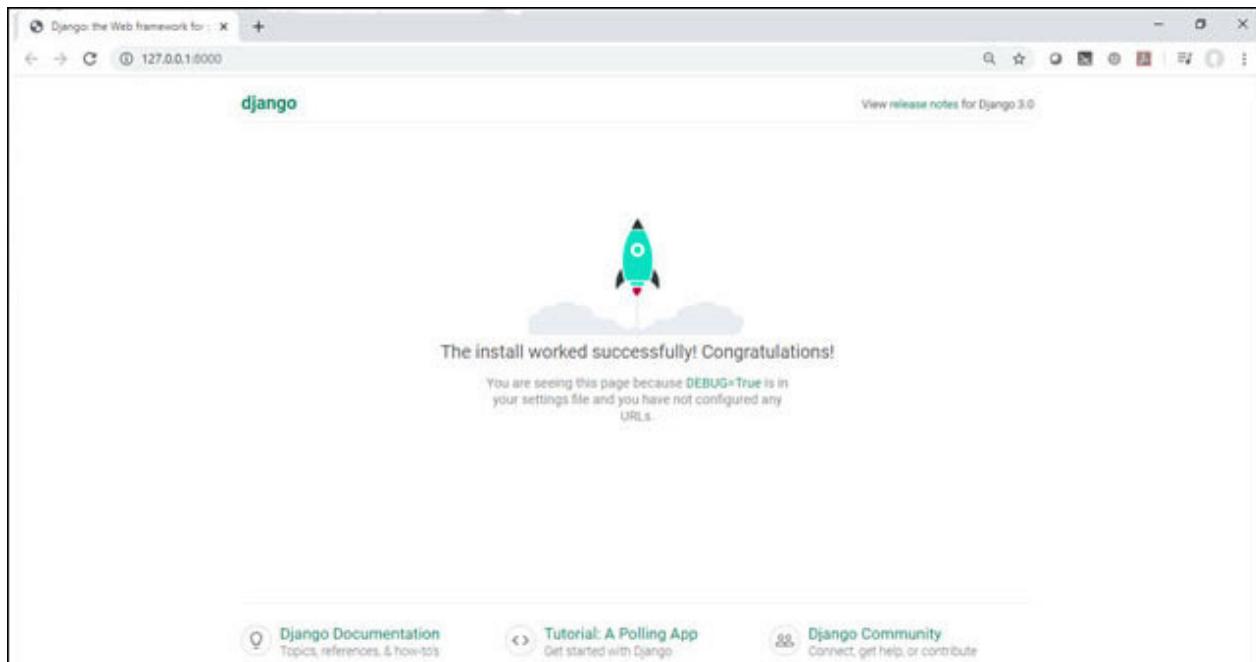


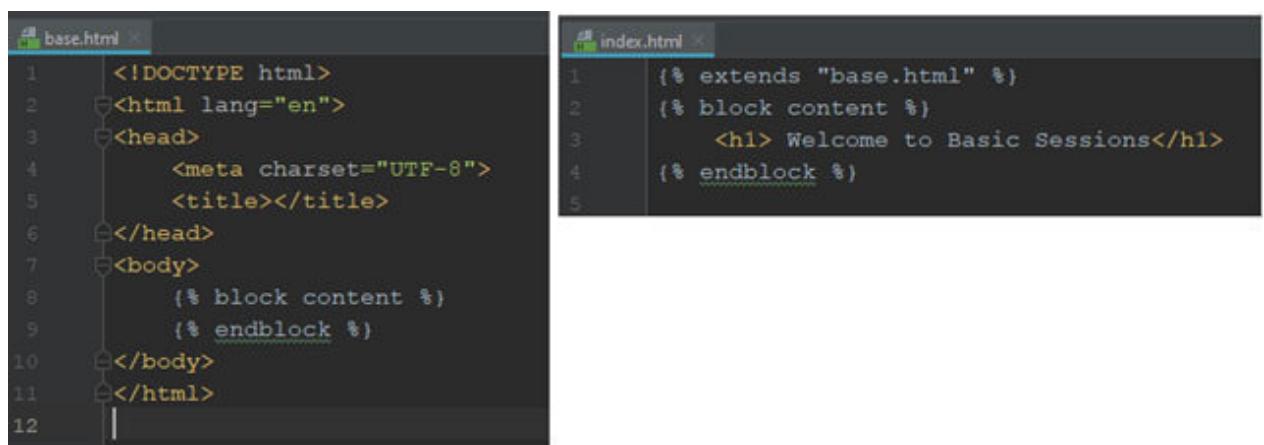
Figure 12.9: The local server

If you have done everything correctly, you will see the preceding screen.

Enhancing the project

Every website has a home page. Let's create one for ours. You have read about templates and template inheritance. This concept will be used now. There are some parts that are common on every webpage of the website. Those sections are kept in

Go to the templates folder of your project and create a homepage for your site index.html and create a base.html file. The base.html file will be inherited by the other templates. For now, we will keep it simple. Type the following code in your files:



The image shows a code editor with two files open side-by-side. On the left is the 'base.html' file, which contains the following code:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title></title>
6  </head>
7  <body>
8      {% block content %}
9      {% endblock %}
10 </body>
11 </html>
```

On the right is the 'index.html' file, which contains the following code:

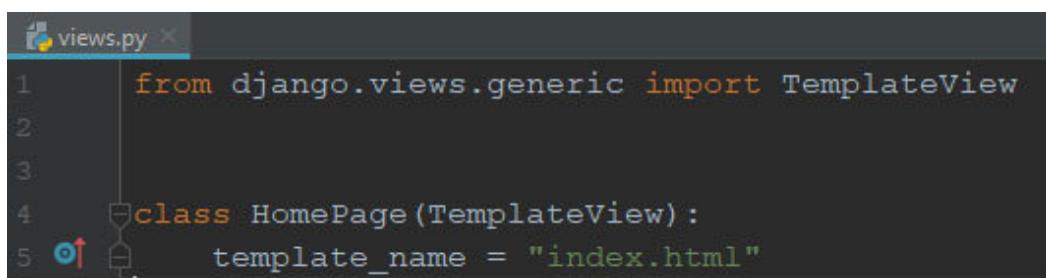
```
1  {% extends "base.html" %}>
2  {% block content %}
3      <h1> Welcome to Basic Sessions</h1>
4  {% endblock %}
```

Figure 12.10: Creating initial base.html and index.html files

In the we have created a content block. The concept of blocks in template inheritance is that whichever file extends base.html will have all the features of If the child file has a block content, then the content block of base.html will be replaced by the content block of the child file.

Thus, the HTML tags and other configurations will be extended to index.html from

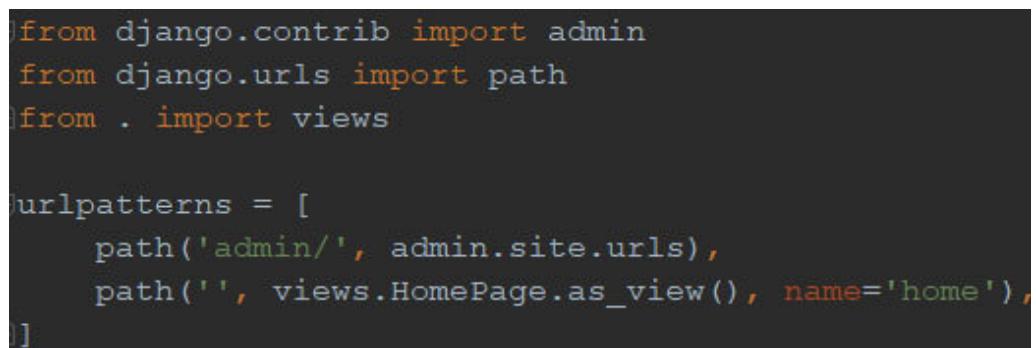
Now, to render the index.html template, we need a view. Go to your folder – \basicsessions-root\basicproject\basicproject\ and create the views.py file. Let's create a simple view that will render the We will use class-based views. From [Chapter 9: Django](#) recall the view used to render a Template – the Update your views.py file as shown in the following screenshot:



```
views.py
1  from django.views.generic import TemplateView
2
3
4  class HomePage(TemplateView):
5      template_name = "index.html"
```

Figure 12.11: Creating the HomePage view of the project

Now, for a request to come to this view, you need to create a url-pattern in the urls.py file. Go to the urls.py file of your project and add the url-pattern as shown in the following screenshot:

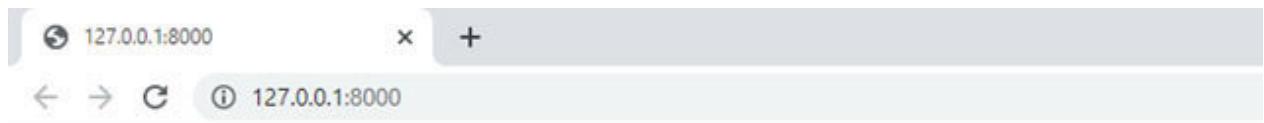


```
urls.py
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.HomePage.as_view(), name='home'),
]
```

Figure 12.12: Url-patterns of the project

Let's test this again. Run the server again and see what shows up on the browser:



Welcome to Basic Sessions

Figure 12.13: The homepage of your project

You should be seeing something like this – the homepage. If you see any error, check the code again. Try stopping the server and running it again.

[Creating superuser](#)

To work further on the website, you must have access to the admin panel of the project. Stop the server and run the python manage.py createsuperuser command:

```
(bsenv) C:\Work\basicsessions-root\basicproject>python manage.py createsuperuser
Username (leave blank to use 'awani'): awanish
Email address: awanish.infarna@gmail.com
Password:
Password (again):
The password is too similar to the email address.
This password is too short. It must contain at least 8 characters.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Figure 12.14: Creating superuser

As seen in the preceding screenshot, Django prompts if the passwords are simple. Since we are on the terminal, I was able to create a simple and short password; if it would have been the user interface of the website, I would not have been able to use such a password. Now, run the server again and log in to the admin panel as an admin using these credentials.

Conclusion

In this chapter, you started your project which you will be deploying in the upcoming chapters. You have set up your project and you are all set to develop the web application now. The setting.py file was updated with correct configurations. You tested all the work by running the server.

In the next chapter, you will work on the account app. This app will introduce you to the Django inbuilt authorization and authentication system. See you there!

Questions

What changes are necessary for a real-world Django project?

Which commands are used to create an app?

How do you register your apps for your project?

How do you create a superuser?

How to log in to the admin app?

CHAPTER 13

The Account App

Introduction

You have your project ready and you are now all set to begin developing your website. Every website has some authentication system using which the admin or users can log in. Also, there is some authorization system using which you can control the users who can access what type of content.

So, to implement these features, we need to work on the authorization and authentication module. This will be covered under the app named account. In this chapter, we will set up a user registration, login, and log out functionalities.

Structure

Difference between authentication and authorization

Django's built-in authentication and authorization module

Models.py

Views.py

Urls.py

Templates

Testing

Conclusion

Objective

The objective of this app is to implement authentication and authorization for the project. To implement user registration, login, and logout, we need to work on the following points:

Our homepage is blank. We need to add signup, login/logout, and home buttons.

A model to store details of users in the database.

A view to implement the user registration functionality, a form where users can fill in their details to register, and a template to display the form.

Similarly, a view to implement the login functionality, a form where users can fill in their username and password, and a template to display the form.

Also, for the logout functionality – a view and a template that would display the success message of logout. We do not need a form for logging out as no data needs to be taken from users.

When a user is logged in, they should be redirected to their profile page or homepage. Since we do not have a user profile

page ready yet, we need to set a redirect to the homepage. This is done in the `settings.py` file.

We will need to create url-patterns for all the views.

Difference between authentication and authorization

Before we move on to code for the account app, we need to understand the difference between the terms, authentication and authorization. You will read about these terms in this chapter.

When users visit any site, let's say Facebook for the first time, they have to register by creating a username, a password, etc. Once registered successfully, they go to the login page and enter their credentials (username and password). The credentials are sent to the server (backend) and checked whether the password and username match. If they match, the user is logged in to the website, and if the credentials do not match, a message is shown to the user that the credentials are incorrect. Then, there is a logout button. When users are done with the website, they click on logout and exit the site. If users wish to visit the site again, they need to log in again. All this falls under the scope of authentication. In a nutshell authentication is user registration, login, and logout.

When you have logged into Facebook, you see posts of some people (like your friends) and do not see posts of some people (like people who are not your friends). You can like and comment on some posts and you cannot like and comment on some posts. This is an authorization. Here, you provide and restrict access to users to perform some activities on your site. It seems complex but it is relatively easy to implement.

Django's built-in authentication module - auth

Django provides a built-in authentication module. You have used this module in your previous project where you used the admin panel. You can consider this module as an app in your project. Like any apps, this app has a model, templates, views, URLs, etc. We will pick and import the required parts of this inbuilt app to our account app and use it for authentication. The User model of the auth module is used to create an authentication system. Let's take a look at the *User*

The User model of the auth module

The User Model is a class in the models.py file located at The User model inherits the AbstractUser class which further inherits the AbstractBaseUser class and PermissionMixin class. You don't need to worry about the AbstractUser or AbstractBaseUser model classes as their properties are present in the User model as we will access those using objects of the User model class itself. So, the fields that the User model provides are as follows:

```
username_validator
username
first_name
last_name
email
is_staff
is_active
date_joined
objects = UserManager()
EMAIL_FIELD = 'email'
USERNAME_FIELD = 'username'
REQUIRED_FIELDS = ['email']
last_login
password
is_active
```

Out of these fields, only the username and password are mandatory fields and the rest are optional. The User model also

inherits a set of methods from the AbstractUser and BaseUser model classes. They are as follows:

clean

get_full_name

get_short_name

email_user

save

get_username

clean

natural_key

is_anonymous

is_authenticated

set_password

check_password

setter

set_unusable_passwo

get_session_auth_hash

get_email_field_name

normalize_username

The PermissionMixin model of the auth module

This is another model defined in the models.py file of the auth module. This model contains the fields and methods necessary to support the Group and Permission. This model is required for authorization. It provides the following fields:

is_superuser
groups

The PermissionMixin model provides the following methods which help in enforcing authorization:

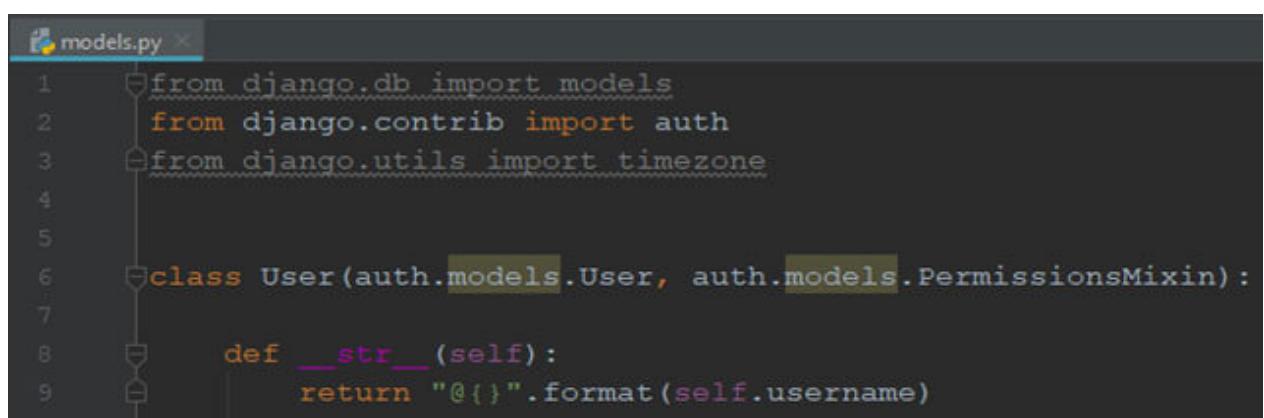
get_user_permissions
get_group_permissions
get_all_permissions
has_perm
has_perms
has_module_perms

The model of your account app will inherit the preceding two models and hence will have the mentioned fields and methods. By using these built-in model classes and their features, we will implement authentication and authorization in our project.

models.py

In order to keep the name resembling to the purpose, we create a model class named User in the models.py file of the account app. As discussed earlier, the User model of the account app will inherit the User model and PermissionMixin model of the auth module. This will create a table in the database with all the fields of the User and PermissionMixin model of the auth module. Their methods will also be available for us to use.

Go to the account app and open the models.py file and update it so that it looks like the following screenshot:



```
models.py
1  from django.db import models
2  from django.contrib import auth
3  from django.utils import timezone
4
5
6  class User(auth.models.User, auth.models.PermissionsMixin):
7
8      def __str__(self):
9          return "@{}".format(self.username)
```

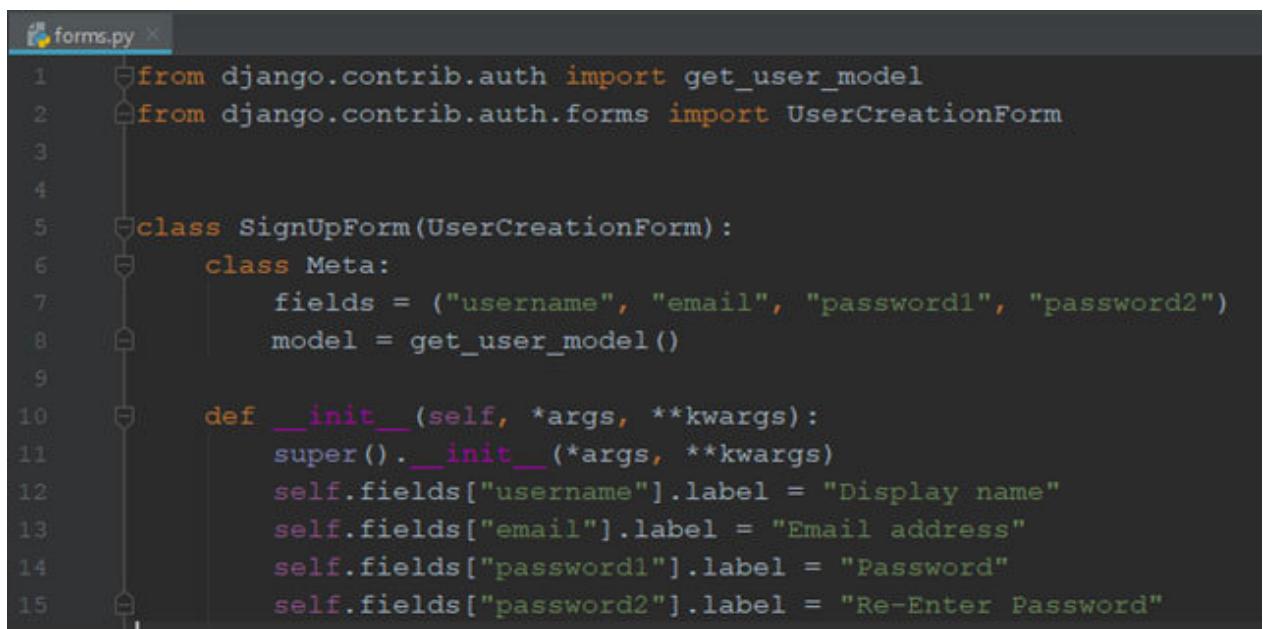
Figure 13.1: models.py file of user app

The `__str__` method is used to get a string representation of the model object. In this case, the `__str__` method will return a string. This string will be used as a link to the user's profile – just like on Instagram or Twitter.

forms.py

From the objectives, we can deduce that we need two forms for the account app: user registration form and login form.

Let us create a user registration form. Django has a built-in form for this purpose named – In the account app folder, create a file forms.py and update it as shown in the following screenshot:



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

class SignUpForm(UserCreationForm):
    class Meta:
        fields = ("username", "email", "password1", "password2")
        model = get_user_model()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields["username"].label = "Display name"
        self.fields["email"].label = "Email address"
        self.fields["password1"].label = "Password"
        self.fields["password2"].label = "Re-Enter Password"
```

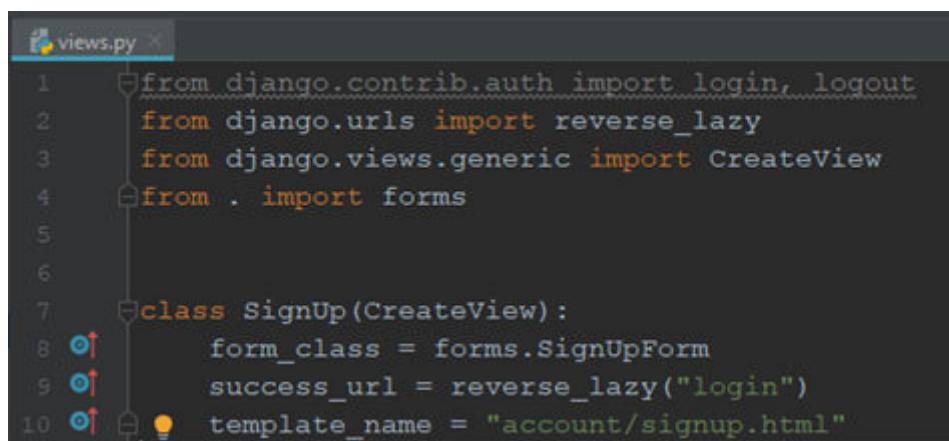
Figure 13.2: The signup form

Here, we are creating a model form from the User model. We choose the username, email, and password2 fields. We will use the inbuilt authentication form (present in the forms.py file of the auth module) for login. So, we do not need to create another form.

views.py

As discussed earlier, we need three views: one for each user registration, login, and logout. Django provides built-in views for login and logout – LoginView and We will be using these built-in views.

The LoginView has the AuthenticationForm defined in itself. Hence, we don't have to worry about the login form. Let us create a view for registration – SignUp view. Update the views.py file of the account app as shown in the following screenshot:



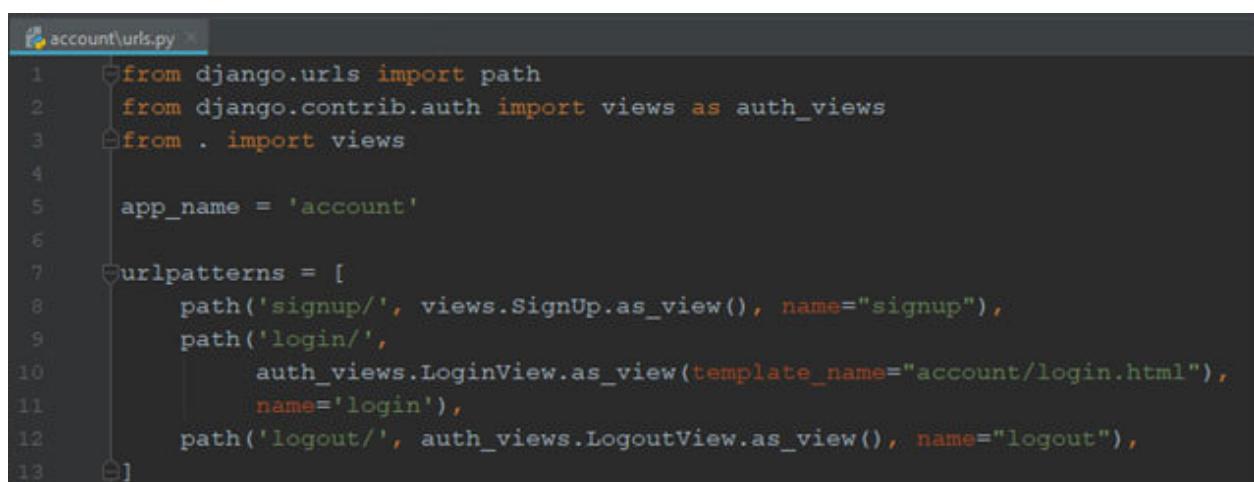
```
views.py
1  from django.contrib.auth import login, logout
2  from django.urls import reverse_lazy
3  from django.views.generic import CreateView
4  from . import forms
5
6
7  class SignUp(CreateView):
8      form_class = forms.SignUpForm
9      success_url = reverse_lazy("login")
10     template_name = "account/signup.html"
```

Figure 13.3: The view.py file of the account app

Here, we created a SignUp view using the built-in generic class-based view – Now, to sign up, the user will need to enter a few details and for that, we will need a form. Hence, we have imported the form.py file in line number 4.

urls.py

To direct the HTTP request to the corresponding views, we need to write the url-patterns. Let us begin with the url-patterns of the account app. Open the urls.py file of the account app and write the following code in it:



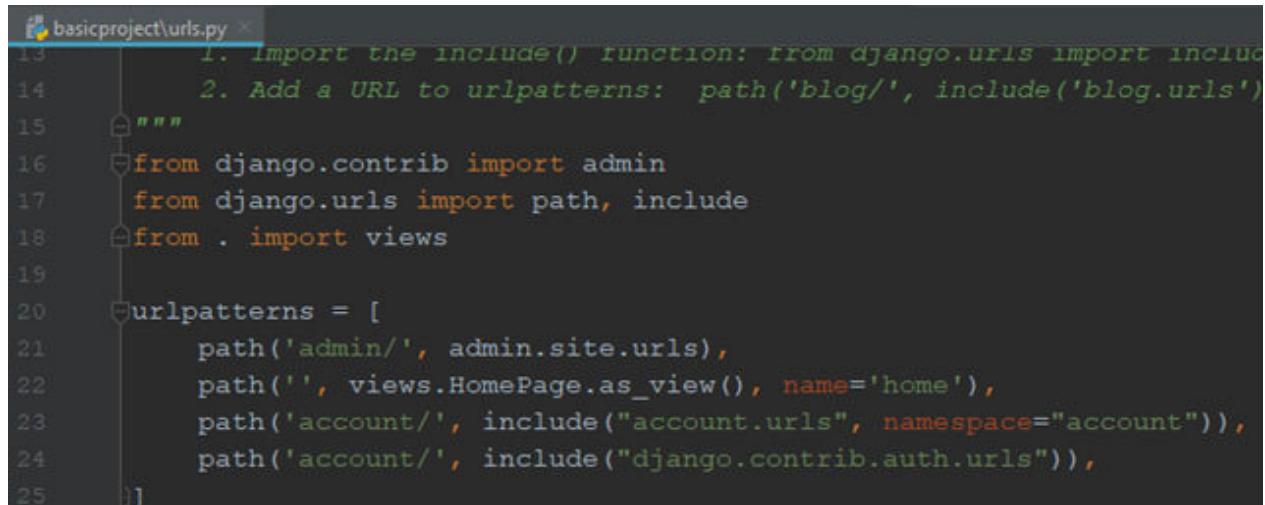
```
account\urls.py
1  from django.urls import path
2  from django.contrib.auth import views as auth_views
3  from . import views
4
5  app_name = 'account'
6
7  urlpatterns = [
8      path('signup/', views.SignUp.as_view(), name="signup"),
9      path('login/',
10          auth_views.LoginView.as_view(template_name="account/login.html"),
11          name='login'),
12      path('logout/', auth_views.LogoutView.as_view(), name="logout"),
13 ]
```

Figure 13.4: Url-patterns of the account app

Here, we have three patterns – login, logout, and signup. The logout and signup patterns are simple URL patterns. In the login url-pattern, you will see an argument with the template being passed to the inbuilt The LoginView by default looks for the registration/login.html template. Since the template we have for the login is we will share this information with the view.

Now, we need to include this url-pattern of the account app into the url-patterns of the project. Open the urls.py file in the

basicproject folder and update it as shown in the following screenshot:



```
basicproject\urls.py
13     1. Import the include() function: from django.urls import include
14     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
15 """
16 from django.contrib import admin
17 from django.urls import path, include
18 from . import views
19
20 urlpatterns = [
21     path('admin/', admin.site.urls),
22     path('', views.HomePage.as_view(), name='home'),
23     path('account/', include("account.urls", namespace="account")),
24     path('account/', include("django.contrib.auth.urls")),
25 ]
```

Figure 13.5: Url-patterns of the project

Here, we have the url-pattern for the admin panel and homepage. We have included the url-patterns from the app we created – Since we are using the built-in auth module, we will have to include the url-patterns in the auth module too.

Remember to keep the url-patterns of the account app before(above) the url-patterns of the built-in auth module. As the auth module will have a lot of url-patterns and if it is mentioned before your app's url-patterns, then your app's url-patterns may not be called as the URLs will match with the auth module's url-patterns and the corresponding view from the auth module will be rendered instead of the account app.

settings.py

By default, the built-in LoginView redirects the user to the /accounts/profile/ page. We do not have this page as of now; therefore, to avoid this error, we need to create the login/logout redirect to some other page. Go to the settings.py file of the project and add these two variables at the end:

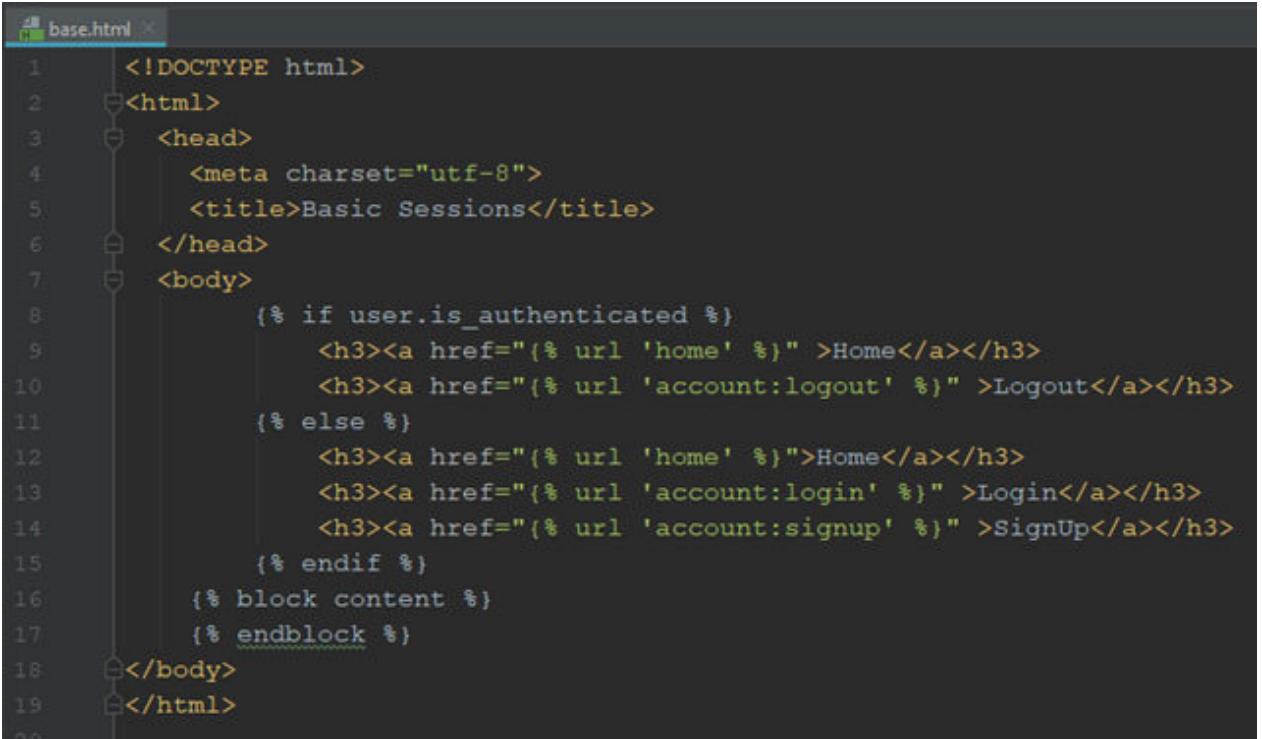
```
124
125     LOGIN_REDIRECT_URL = "home"
126     LOGOUT_REDIRECT_URL = "home"
127 |
```

Figure 13.6: Adding login and logout redirect URLs in the settings.py file

Now, whenever any user will login or logout, they will be redirected to the url-pattern with the name home which is the homepage

Templates

Firstly, we need to add the basic links to our homepage like login, logout, signup, and home. Open the base.html in basicproject/templates/base.html and update it as shown in the following screenshot:



The screenshot shows a code editor window with the file 'base.html' open. The code is a Django template. It starts with the DOCTYPE declaration and an HTML tag. Inside the HTML tag, there is a head section containing a meta tag for utf-8 encoding and a title tag with the text 'Basic Sessions'. The body section contains conditional logic. If the user is authenticated, it displays three links: 'Home', 'Logout', and 'SignUp'. If the user is not authenticated, it displays three links: 'Home', 'Login', and 'SignUp'. The code uses Django's templating syntax, including if statements and block tags.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Basic Sessions</title>
    </head>
    <body>
        {% if user.is_authenticated %}
            <h3><a href="{% url 'home' %}">Home</a></h3>
            <h3><a href="{% url 'account:logout' %}">Logout</a></h3>
        {% else %}
            <h3><a href="{% url 'home' %}">Home</a></h3>
            <h3><a href="{% url 'account:login' %}">Login</a></h3>
            <h3><a href="{% url 'account:signup' %}">SignUp</a></h3>
        {% endif %}
        {% block content %}
        {% endblock %}
    </body>
</html>
```

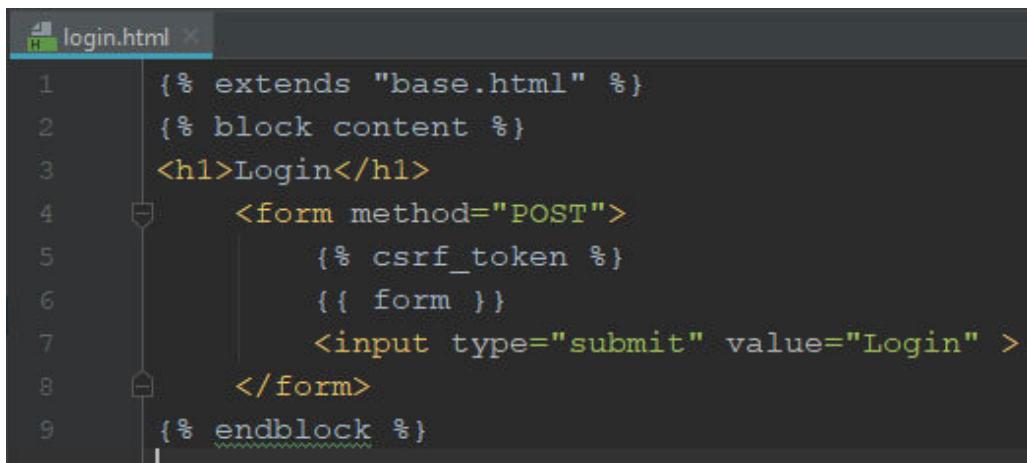
Figure 13.7: base.html file of the project

Here, we have added a check to see whether the user is logged in or not. We can check this using the `is_authenticated` method of the User model of the auth module. Our User model of the account app has inherited all those methods, thus we can use these methods using objects of our model. If the user is authenticated,

the page should show the logout button and home button. If the user is not logged in the home page, login and signup buttons should show up.

Notice how the links are being made. We are using the name of the url-patterns along with the url template tag to create links on the webpage. If the url-pattern is in any app, the name of the app should also be mentioned. Then, we have our content block which will be replaced by the content block of other templates which inherit this base.html template.

We already have the index.html template. Let us create the login.html template. Go to basicproject/account/templates/account/ and create a file named Write the following code in the file:



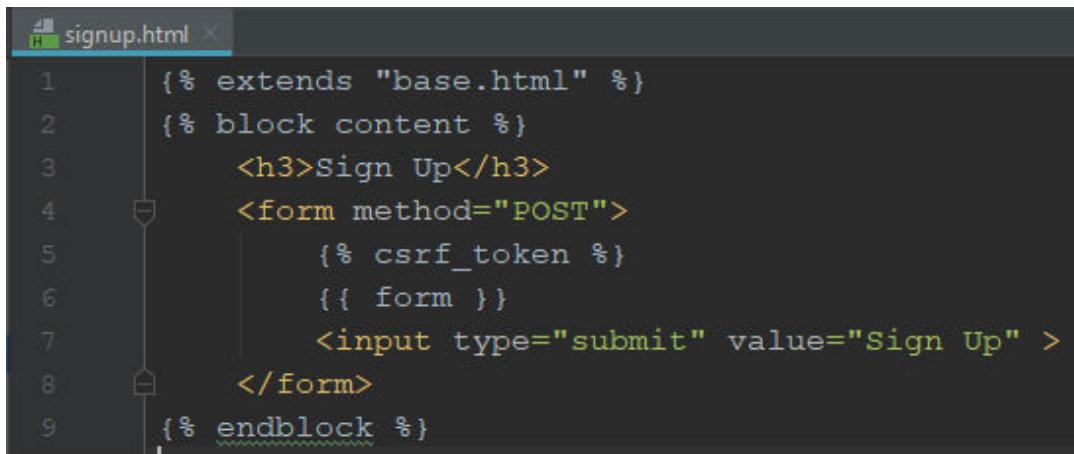
```
login.html ×
1  {% extends "base.html" %} 
2  {% block content %} 
3      <h1>Login</h1> 
4      <form method="POST"> 
5          {% csrf_token %} 
6          {{ form }} 
7          <input type="submit" value="Login" > 
8      </form> 
9  {% endblock %}
```

Figure 13.8: login.html file of the account app

This is considerably a basic form template. It extends the base.html you created earlier. Then, the block content begins. We have the csrf_token and then, the form variable. You may not recall sending any form to this variable as you did not create any form. This form

variable is fed by the built-in LoginView which uses the built-in AuthenticationForm. Then, there is the login button. Once the user enters the username and password and clicks on the **Login** button, it will invoke some code written in the LoginView, and based on the credentials, the user will either be logged in or an incorrect credential message will be displayed. All this work is done by the inbuilt We did not have to write a single line of code for authentication. This is the benefit of using built-in features.

Let's create the SignUp template. Go to basicproject/account/templates/account/ and create a file named Write the following code in the file:



```
signup.html
1  {% extends "base.html" %} 
2  {% block content %} 
3      <h3>Sign Up</h3> 
4      <form method="POST"> 
5          {% csrf_token %} 
6          {{ form }} 
7          <input type="submit" value="Sign Up" > 
8      </form> 
9  {% endblock %}
```

Figure 13.9: signup.html file of the account app

This again is a simple form template. Here, we have extended the base.html template. Then, we have our content block. Then, we have our form variable. This form variable is provided by the SignUp view we created which used the This template will display the form. Then, we have the submit button which when clicked will invoke some code in the inbuilt UserCreationForm which is

inherited by the SignUpForm and the user details be saved in the database.

We have all our templates ready. It's now time to test.

Testing

To test the app, run the following commands:

```
python manage.py makemigrations account  
python manage.py migrate  
python manage.py runserver
```

If you have not made any error, you will be able to see the following screen.

Homepage:



Figure 13.10: Home page of the project

When you click on the SignUp link, as shown in the preceding screenshot, you will see the following page with the signup form

The screenshot shows a web browser window with the title 'Basic Sessions'. The address bar displays the URL '127.0.0.1:8000/account/signup/'. The page content includes navigation links ('Home', 'Login', 'SignUp') and a 'Sign Up' section. In the 'Sign Up' section, there is a 'Display name' input field containing 'ram', a note stating 'Required. 150 characters or fewer. Letters, digits and @/./-/_. only.', an 'Email address' input field containing 'ram@gmail.com', a 'Password' input field containing '*****', and a 'Re-Enter Password' input field also containing '*****'. Below these fields is a note 'Enter the same password as before, for verification.' and a 'Sign Up' button.

Figure 13.11: Signup page

Once the form has been filled, you can click on the **Signup** button. If the details are entered correctly, the user will be registered, and you will see the login page as shown in the following screenshot

The screenshot shows a web browser window with the title 'Basic Sessions'. The address bar displays the URL '127.0.0.1:8000/account/login/'. The page content includes navigation links ('Home', 'Login', 'SignUp') and a 'Login' section. In the 'Login' section, there are 'Username' and 'Password' input fields, both containing 'ram', and a 'Login' button.

Figure 13.12: Login page

Now, you can log in with the credential created. Enter the user name and password and click on login and you will see the following screenshot

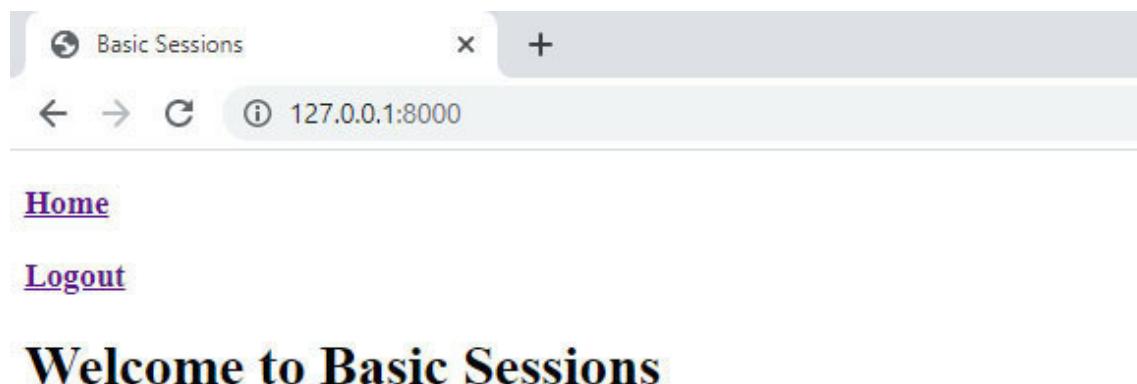


Figure 13.13: Home page after logging in

This is the homepage after login. You may click on logout and you will be back on the homepage before logging in as shown in the following screenshot

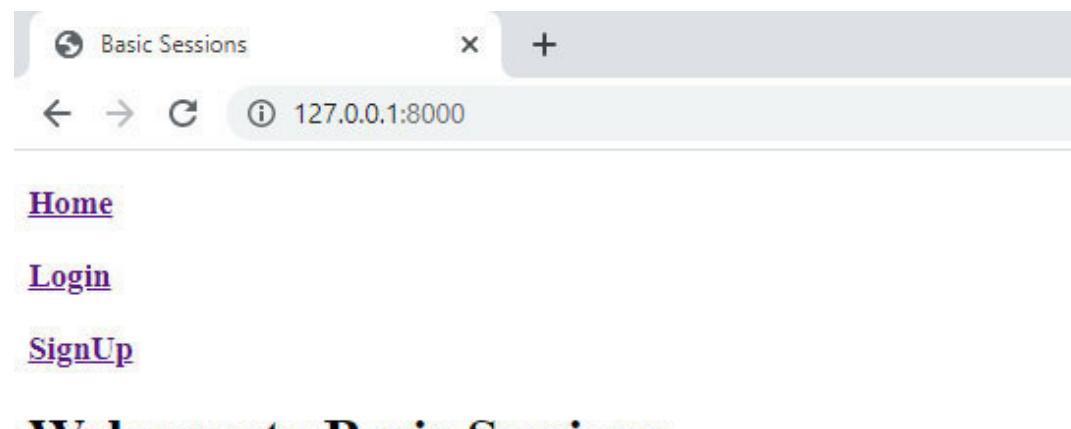


Figure 13.14: After logging out

If all the links work fine, it means you have completed the account app on your website. If you see any errors, try to debug it by reading the error messages and figuring out the file and line of code which has resulted in the error. If you have followed the steps correctly, you may not see any error.

Create at least 10-15 users as we will need some users on our website to implement the apps in the next chapters. Log in to the admin panel and check.

This implementation was not difficult as we used a lot of built-in code. This is the standard way of programming. It keeps your code error-free as the built-in code is rigorously tested against a huge number of test cases and by many people. You may be clueless about how this built-in code works. It's easy to figure out that.

Open your project in PyCharm, press Ctrl, and hover over the built-in classes/functions/modules used in your project and they will get highlighted. Click on it and you will reach its code. In this way, you can read the built-in code and see how it is working. Some might say the webpages are looking very ugly. Styling the website is done by using CSS, bootstrap, JavaScript, etc. These are written in the templates. If we write the styling code pieces now, the templates will look complex and it is important that you focus on the Django parts of the template and not on CSS and JavaScript parts. Once the website is completed, we will style the website using Bootstrap.

Conclusion

This was the account app that was responsible for authentication and authorization. We worked on the models.py file and set up a model that will be used to store user information. We created forms which would help us getting data from users and entering it in the database. We created our login, logout, and homepage views. We wrote the url-patterns and templates to handle requests and display data on-site.

In the next chapter, you will be working on the second app – Genre. This app will be responsible for the segregation of the post. The different genres will have different kinds of posts. The structure will be more or less the same as this app.

Questions

What is the difference between authentication and authorization?

Which model is used to implement the built-in auth app?

What are the fields present in the auth app user model?

How do you inherit a model of different modules in your app?

Do you need to register the inherited model in the admin app?

How to get the user model currently being used in a project?

CHAPTER 14

The Genre App

Introduction

Your website will have content from many users. All these users will have different areas of interests and they will post content of different genres and would like to read posts on different topics. Thus, we need a system that would segregate the content based on its type or genre.

In this chapter, we will give our users options to associate themselves with genres. We create a list of genres and users can associate themselves with the genre. Also, users can create new genres and ask their friends to associate themselves with the genre.

Structure

models.py

admin.py

views.py

urls.py

Templates

Testing

Conclusion

Objective

The objective of this app is to add Genres to the website. We need to work on the following points:

The homepage should have a link to see the list of Genres. Here, unregistered users would be able to see the list of Genres present on the website. On clicking on any genre, they should be able to read the posts.

A model to store details of Genres in the database.

A model to keep track of the users associated with which Genres.

A view to implement the genre functionality, a form where users can fill in their details to create the genre, and a template to display the form.

A view that enables users to associate/disassociate to any genre. We will need to add the associate/disassociate button to several templates.

A view to display a list of genres present on the website. A template to display the list.

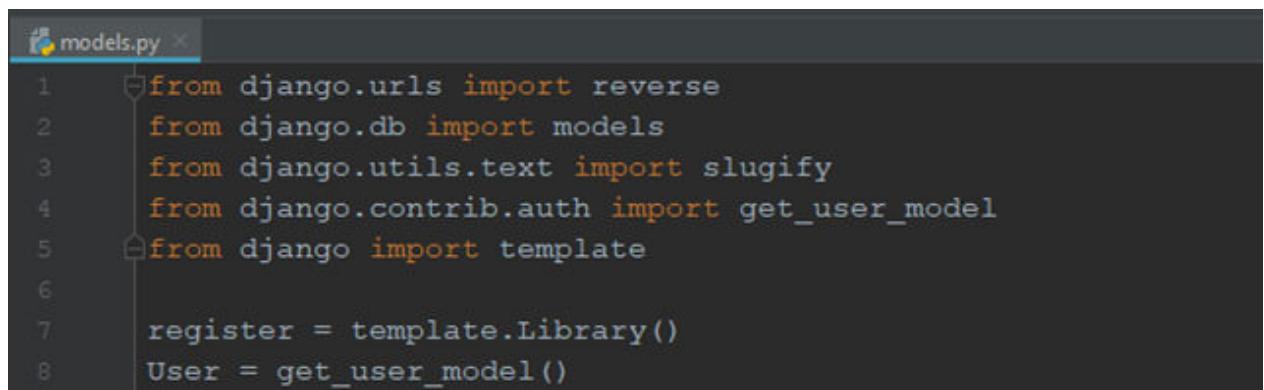
A view to implement a detail view for a single genre. A template to show the genre.

This is relatively a simple app. Let us begin.

models.py

To enable the setup of different genres, we will need a model which will contain information about the model which will be later used to link posts to specific genres.

Go to the basicproject/genre/ and in write the following code:



```
models.py
1  from django.urls import reverse
2  from django.db import models
3  from django.utils.text import slugify
4  from django.contrib.auth import get_user_model
5  from django import template
6
7  register = template.Library()
8  User = get_user_model()
```

Figure 14.1: Import to use in the models.py file

Let us check out what the preceding imports do:

reverse: If we want to dynamically create URLs on templates which should point to the object of a model, we need to use this reverse function. Just create a method in the model class `get_absolute_url` and call this reverse function in return. The syntax is as follows:

```
reverse(view_function_name, **kwargs) // for function-based views
reverse('appname:url-patternname', **kwargs) // for class-based views
```

It creates slugs.

get_user_model: As clear from the name, this function gets the model which is being used to handle the user. Then, by calling this model in line number 8, we get access to the attributes and method of the User model. In our website, since the users are handled in the account app and model is we will get access to this model via User created in line number 9.

template: This module helps in creating custom template tags.

Now, let's create the Model classes. Add the following model class to your models.py file:

```
11  class Genre(models.Model):
12      name = models.CharField(max_length=255, unique=True)
13      slug = models.SlugField(allow_unicode=True, unique=True)
14      description = models.TextField(blank=True, default='')
15      description_html = models.TextField(editable=False, default='', blank=True)
16      fellows = models.ManyToManyField(User, through="GenreFellow")
17
18  @modelона
19  def __str__(self):
20      return self.name
21
22  def save(self, *args, **kwargs):
23      self.slug = slugify(self.name)
24      self.description_html = self.description
25      super().save(*args, **kwargs)
26
27  def get_absolute_url(self):
28      return reverse('genre:single', kwargs={"slug": self.slug})
29
30  class Meta:
31      ordering = ["name"]
```

Figure 14.2: The Genre model class

In this model class, we have a few fields and a few methods. All the fields are simple. The fellows are a ManyToMany field that is implemented via the GenerFellow model class. This is to keep track of the users associated with a genre.

The get_absolute_url method is used to create URLs for the Genre class objects.

We need one more table in the database which will save information about which user is associated with which genres. Add the following model class in your models.py file:

```
33  class GenerFellow(models.Model):
34      genre = models.ForeignKey(Genre, related_name="fellowships", on_delete=models.CASCADE)
35      user = models.ForeignKey(User, related_name='user_genre', on_delete=models.CASCADE)
36
37      def __str__(self):
38          return self.user.username
39
40      class Meta:
41          unique_together = ("genre", "user")
```

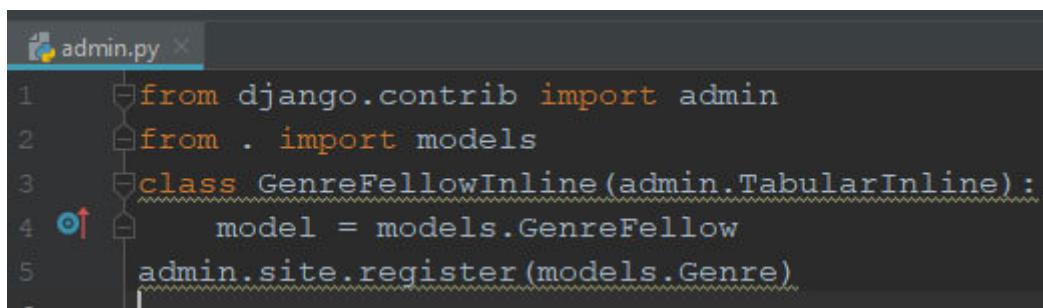
Figure 14.3: The GenerFellow model class

In this model class, we have two fields: genre and user. Whenever a user associates with any genre, a record is added to this table.

admin.py

In the account app, we did not create any new model but simply used a model already available. Here, we have created our models and thus, we need to register the models to the admin.py file to make them available on the admin panel.

Open the basicproject/genre/admin.py file and update it as shown in the following screenshot:



```
admin.py
1 from django.contrib import admin
2 from . import models
3 class GenreFellowInline(admin.TabularInline):
4     model = models.GenreFellow
5     admin.site.register(models.Genre)
```

A screenshot of a code editor showing the 'admin.py' file. The file contains Python code for registering models with the Django admin interface. It includes imports for 'admin' from 'django.contrib' and 'models' from the current directory. A class 'GenreFellowInline' is defined as a TabularInline for the 'GenreFellow' model. Finally, 'admin.site.register(models.Genre)' is called to register the 'Genre' model. A red arrow points to the first line of code, highlighting the 'from django.contrib import admin' statement.

Figure 14.4: The admin.py file

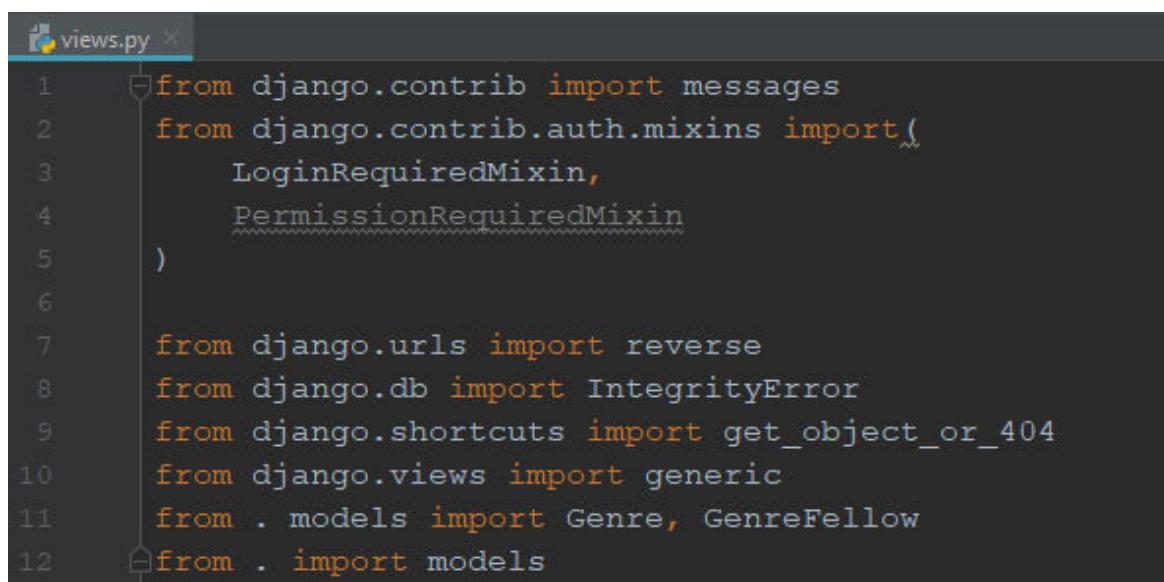
Here, we have two models – Genre and The GenreFellow model is kind of dependent on the Genre model. We can't have a record in the GenreFellow model unless we have that Genre in the Genre model.

So, in a situation where you are adding genres to the website using the admin panel, it would be convenient to be able to add the members to the genre on the same screen. To get access to

both the models on the same page on the admin panel, we will use the `admin.TabularInline` class.

views.py

We have our models ready. Now, let's create the views. Open basicproject/genre/views.py and add the following code in it:



```
views.py
1  from django.contrib import messages
2  from django.contrib.auth.mixins import (
3      LoginRequiredMixin,
4      PermissionRequiredMixin
5  )
6
7  from django.urls import reverse
8  from django.db import IntegrityError
9  from django.shortcuts import get_object_or_404
10 from django.views import generic
11 from .models import Genre, GenreFellow
12 from . import models
```

Figure 14.5: The imports for views of the genre app

Let us take a look at each of these imports:

These are display messages on the screen from the view. You don't have to work on templates if you want to display any message on the screen.

Every view that needs the user to be logged in will need to inherit this class. This is how we implement authorization.

To create URLs.

This is a database exception and is invoked when any of the database constraint is violated. In the `GenreFellow` model class, we added a constraint of If a user tries to associate to any genre twice, this will be invoked.

This calls the `get()` method of the given model manager and if the model is not found, it returns a 404 page (page not found error page).

Now, add the following views to your `views.py` file:

```
15  class CreateGenre(LoginRequiredMixin, generic.CreateView):
16      fields = ("name", "description")
17      model = Genre
18
19
20  class SingleGenre(generic.DetailView):
21      model = Genre
22
23
24  class ListGenre(generic.ListView):
25      model = Genre
```

Figure 14.6: *CreateGenre, SingleGenre and ListGenre view classes of the genre app*

These are pretty simple views:

This view is used to create a new `Genre`. It uses the `generic` This is the same as the one we used in the `account` app views – This view inherits `LoginRequiredMixin` which ensures the user necessary

to be logged in to create a view. Since we are using a built-in view here, we need to provide the model name and the fields of that model on which this view will act. Hence, the two variables – fields and model.

SingleGenre: This again implements a built-in view – It will display the details of the object of the model given. Here, we have given the model This view will show us the details of any

This view is responsible to display a list of all the Genres present in the Genre model.

Now, we need a view to implement the association of users to any genre. To keep it simple, let us call it the JoinGenre view. Add the following view in your view.py file:

```
28 class JoinGenre(LoginRequiredMixin, generic.RedirectView):
29
30     def get_redirect_url(self, *args, **kwargs):
31         return reverse("genre:single",
32                         kwargs={"slug": self.kwargs.get("slug")})
33
34     def get(self, request, *args, **kwargs):
35         genre = get_object_or_404(Genre,
36                                 slug=self.kwargs.get("slug"))
37         try:
38             GenreFellow.objects.create(user=self.request.user,
39                                         genre=genre)
40         except IntegrityError:
41             messages.warning(self.request,
42                             ("Warning, already a fellow of {}".format(genre.name)))
43         else:
44             messages.success(self.request,
45                             "You are now a fellow of the {} genre.".format(genre.name))
46         return super().get(request, *args, **kwargs)
```

Figure 14.7: JoinGenre view class

The JoinGenre view class inherits two classes – LoginRequiredMixin and You know what LoginRequiredMixin does. RedirectView is used to redirect to another page. Since we do need a template specific to this view, we will have to **Redirect** to some other page. When you click on the **Associate** button, the expectation is that you are associated with the Once the association is done, you can either redirect the user to the same page or any other page. Here, we choose to redirect the user to the Genre detail page.

We have two methods in this view:

This method overrides the default method present in the It obtains the URL of the detail page of the Genre the user has associated to.

This overrides the get() method of the It performs the association process. Firstly, we get the object of the model class in use by calling get_object_or_404 and passing the model name and the slug as arguments.

Once we have the genre the user wants to associate to, we will try to create a record in the GenreFellow model class using the user and genre. Here, we expect the IntegrityError exception and use the message module imported earlier to display the message that the user is already associated with the genre. If the exception does not occur, a success message is displayed.

We need a disassociation view too. Let us call it LeaveGenre view. Add the following view to your views.py file:

```
48     class LeaveGenre(LoginRequiredMixin, generic.RedirectView):
49
50     def get_redirect_url(self, *args, **kwargs):
51         return reverse("genre:single", kwargs={"slug": self.kwargs.get("slug")})
52
53     def get(self, request, *args, **kwargs):
54         try:
55             fellowship = models.GenreFellow.objects.filter(
56                 user=self.request.user,
57                 genre__slug=self.kwargs.get("slug")
58             ).get()
59         except models.GenreFellow.DoesNotExist:
60             messages.warning(
61                 self.request,
62                 "You can't leave this genre because you aren't in it."
63             )
64         else:
65             fellowship.delete()
66             messages.success(
67                 self.request,
68                 "You have successfully left this genre."
69             )
70     return super().get(request, *args, **kwargs)
```

Figure 14.8: LeaveGenre view class

Like the JoinGenre view, this also inherits the LoginRequiredMixin and RedirectView for the same purpose. Here, in the get method, first, we find out the record in the GenreFellow model which is responsible for the association of the user with the genre. If the record is found, we delete that record, and the user is disassociated from the genre. If the record is not found, the DoesNotExist error is raised and caught in the except block. A message is displayed that the user is not associated with the genre chosen for disassociation.

[urls.py](#)

So, we have our views and models ready. Now, let us connect the requests to the corresponding views using the url-patterns. Firstly, in the urls.py file of your project – add a url-pattern to include the url-patterns of the genre app:

```
20     urlpatterns = [
21         path('admin/', admin.site.urls),
22         path('', views.HomePage.as_view(), name='home'),
23         path('account/', include("account.urls", namespace="account")),
24         path('account/', include("django.contrib.auth.urls")),
25         path('genre/', include("genre.urls", namespace="genre")),
26     ]
```

Figure 14.9: Url-patterns of the genre the project

Now, in the basicproject/genre/ folder, create a urls.py file and update it as shown in the following screenshot:

```
genre\urls.py ×
1  from django.urls import path
2  from . import views
3
4  app_name = 'genre'
5
6  urlpatterns = [
7      path('', views.ListGenre.as_view(), name="all"),
8      path('new/', views.CreateGenre.as_view(), name="create"),
9      path('<slug:slug>/', views.SingleGenre.as_view(), name='single'),
10     path('associate/<slug:slug>/', views.JoinGenre.as_view(), name="join"),
11     path('disassociate/<slug:slug>/', views.LeaveGenre.as_view(), name="leave"),
12 ]
```

Figure 14.10: Url-pattern of the genre app

We have the url-patterns named as follows:

all: This is to display all the genres present.

create: This is to create a new genre.

single: This is to display a single genre in the detail view.

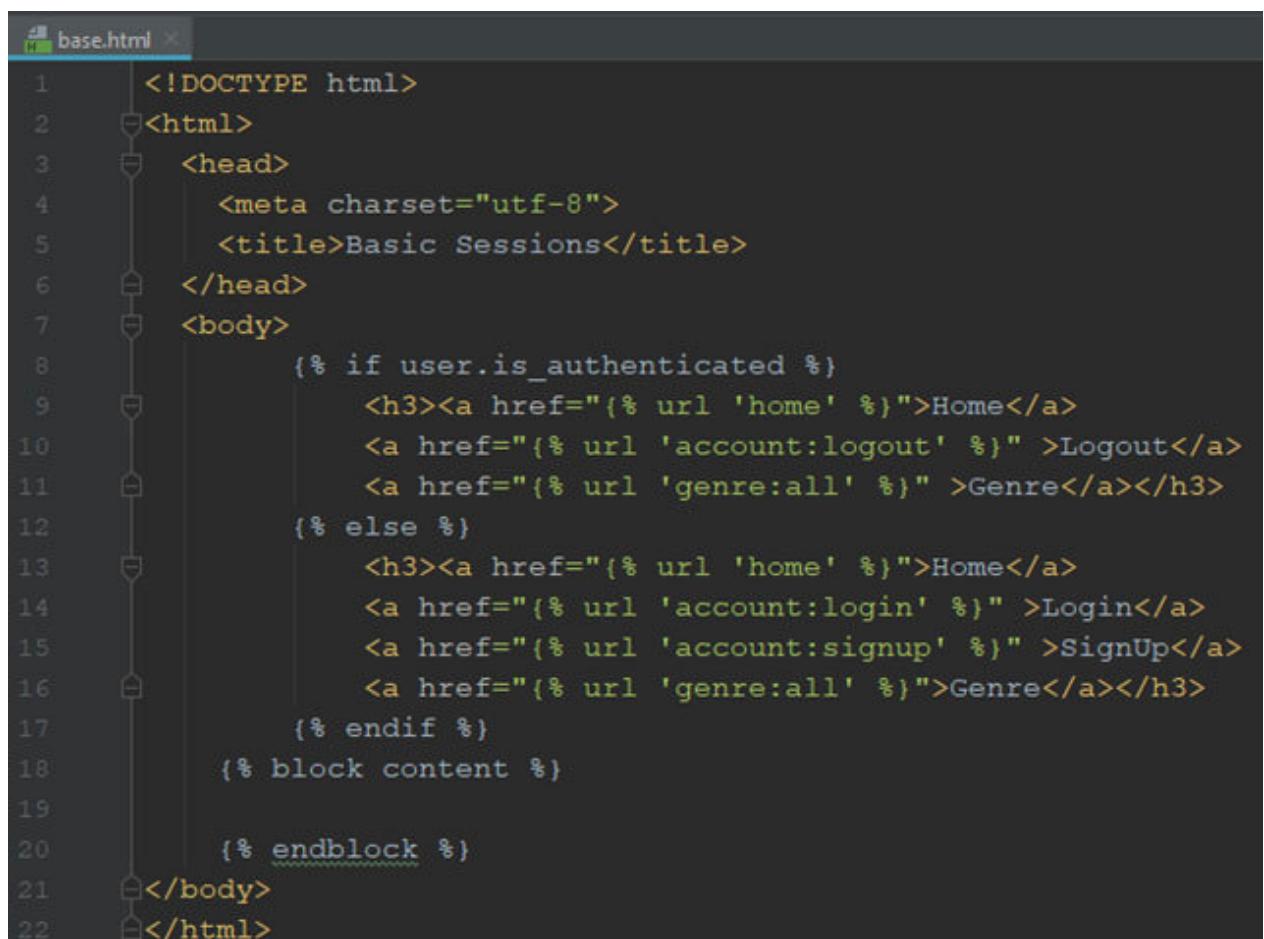
join: This is to associate to any genre.

leave: This is to disassociate from any genre.

So, our urls.py files are ready. Now, we need templates to display the pages.

Templates

We have a genre app almost ready. So, let's add genre links to the homepage of our website. Got to basicproject/templates/base.html and update it as shown in the following screenshot:



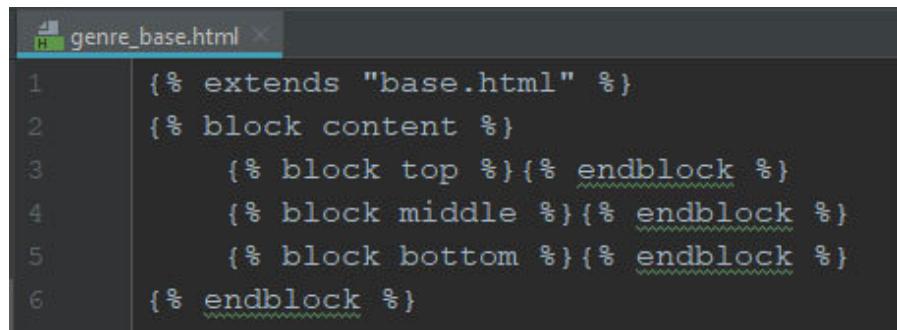
The screenshot shows a code editor window with the file 'base.html' open. The code is a Django template. It starts with the standard HTML doctype and structure. Inside the body, there is a conditional block that checks if the user is authenticated. If true, it displays three links: 'Home', 'Logout', and 'Genre'. If false, it displays four links: 'Home', 'Login', 'SignUp', and 'Genre'. After the conditional block, there is a block for 'content' and an endblock marker. The code ends with closing tags for body and html.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Basic Sessions</title>
    </head>
    <body>
        {% if user.is_authenticated %}
            <h3><a href="{% url 'home' %}">Home</a>
            <a href="{% url 'account:logout' %}">Logout</a>
            <a href="{% url 'genre:all' %}">Genre</a></h3>
        {% else %}
            <h3><a href="{% url 'home' %}">Home</a>
            <a href="{% url 'account:login' %}">Login</a>
            <a href="{% url 'account:signup' %}">SignUp</a>
            <a href="{% url 'genre:all' %}">Genre</a></h3>
        {% endif %}
        {% block content %}
        {% endblock %}
    </body>
</html>
```

Figure 14.11: Updated base.html of the project

Here, just like the links of the account app, we have added links of the genre app. Now, go to the basicproject/genre/templates/genre/

folder and create a genre_base.html file. Write the following code in this file:



A screenshot of a code editor window titled "genre_base.html". The code is a Python template with the following content:

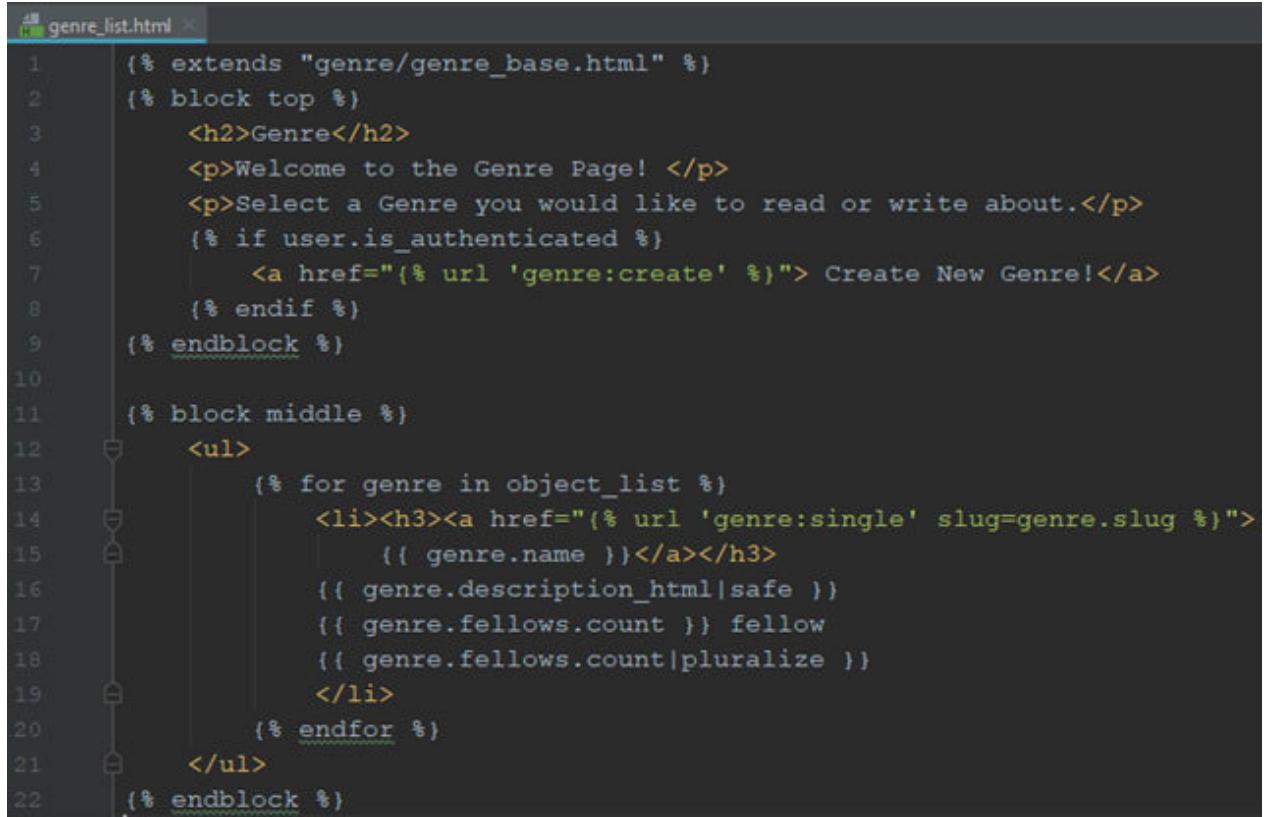
```
1  {% extends "base.html" %}\n2  {% block content %}\n3      {% block top %}{% endblock %}\n4      {% block middle %}{% endblock %}\n5      {% block bottom %}{% endblock %}\n6  {% endblock %}
```

Figure 14.12: genre_base.html file

So, this genre_base.html file extends the base.html file of the project. This solves the need of adding everything of base.html (except the content block) to our Further, we have created three sections in the content block of We needed three blocks so that we can keep the genre headings in the top section, basic details of the genre in the middle section, and posts of the genre in the bottom section.

So, now the other templates of the genre app will extend Since genre_base.html already extends the other templates won't need to extend base.html.

Looking at the objectives, we need a template to display all the genres. We just add a link to this template in the base.html file. So, in the same folder of create a genre_list.html file and update it as shown in the following screenshot:



```
genre_list.html
1  {% extends "genre/genre_base.html" %} 
2  {% block top %} 
3      <h2>Genre</h2> 
4      <p>Welcome to the Genre Page! </p> 
5      <p>Select a Genre you would like to read or write about.</p> 
6      {% if user.is_authenticated %} 
7          <a href="{% url 'genre:create' %}"> Create New Genre!</a> 
8      {% endif %} 
9      {% endblock %} 
10 
11     {% block middle %} 
12         <ul> 
13             {% for genre in object_list %} 
14                 <li><h3><a href="{% url 'genre:single' slug=genre.slug %}"> 
15                     {{ genre.name }}</a></h3> 
16                     {{ genre.description_html|safe }} 
17                     {{ genre.fellows.count }} fellow 
18                     {{ genre.fellows.count|pluralize }} 
19                 </li> 
20             {% endfor %} 
21         </ul> 
22     {% endblock %}
```

Figure 14.13: *genre_list.html* file

This is a king of the genre app home page. Here, we have a link to create the genre at the top. In the middle, we have the list of genres present with a little description. Logged in users can create groups. The unauthenticated users can browse through the genres present.

Since we are displaying a list of genres, users would like to see them. For that, we need a template – create a file named *genre_detail.html* in the same folder and update it as shown in the following screenshot:

```
genre_detail.html
1  {% extends "genre/genre_base.html" %}

2
3  {% block top %}
4      <h1>{{genre.name}}</h1>
5      <h2> Fellow Count: {{genre.fellows.count}}</h2>
6      {% if user.is_authenticated %}
7          {% if user in genre.fellows.all %}
8              <a href="{% url 'genre:leave' slug=genre.slug %}">Disassociate </a>
9          {% else %}
10             <a href="{% url 'genre:join' slug=genre.slug %}">Associate</a>
11         {% endif %}
12     {% else %}
13         <p>Login to associate to this genre.</p>
14     {% endif %}
15  {% endblock %}
16  {% block middle %}
17      <h2>No posts in this genre yet!</h2>
18  {% endblock %}
```

Figure 14.14: *genre_detail.html* file

Here, we have a little description in the top section. We have the **Associate/Disassociate** button. In the middle section, we will have a list of posts for the genre. Since we do not have any posts as of now, we just have a message.

We also need a template to render the form to create these genres. Create a *genre_form.html* file and update it as shown in the following screenshot:

```
genre_form.html
1  {% extends "genre/genre_base.html" %}

2
3  {% block middle %}
4      <h4>Create A New Genre</h4>
5      <form method="POST" action="{% url 'genre:create' %}" id="genreForm">
6          {% csrf_token %}
7          {{ form }}
8          <input type="submit" value="Create" >
9      </form>
10  {% endblock %}
```

Figure 14.15: *genre_form.html*

Here, we created the form by using the CreateGenre view in the views.py file. If you look at the CreateGenre class, we have two attributes: fields and a model. The model is Genre and fields are name and description. Thus, the form will have these two fields and the data entered will be saved to the Genre model class' database table.

All our templates are ready now. Time to Test!

Testing

To test the application, run the following commands:

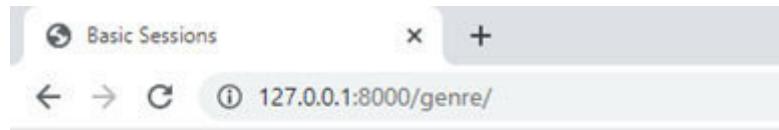
```
python manage.py makemigrations genre  
python manage.py migrate  
python manage.py runserver
```

If you have not made any error, you will be able to see the following screen:



Figure 14.16: Home page before login

We update our homepage a bit. We have the **Genre** link here. Let's see what we get while using the site without logging in. Click on



[Home](#) [Login](#) [SignUp](#) [Genre](#)

Genre

Welcome to the Genre Page!

Select a Genre you would like to read or write about.

- [HTML](#)

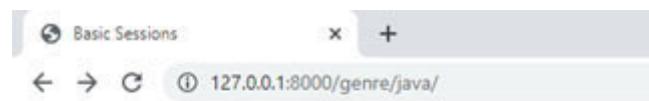
For webpage cretero 1 fellow

- [Java](#)

For Java fans 1 fellow

Figure 14.17: Genre page before login

Here, I have already created a few Genres to test. Let's click on



[Home](#) [Login](#) [SignUp](#) [Genre](#)

Java

Fellow Count: 1

Login to associate to this genre.

No posts in this genre yet!

Figure 14.18: Genre detail page before login

So, this is how the **Genre** app works without logging in. It works fine but looks ugly. Again, we will style it once the website is

completed. Now, let's log in. Hope you remember the credentials of the user created in the previous chapter. If not, sign up again:



Figure 14.19: Home page after login

Click on

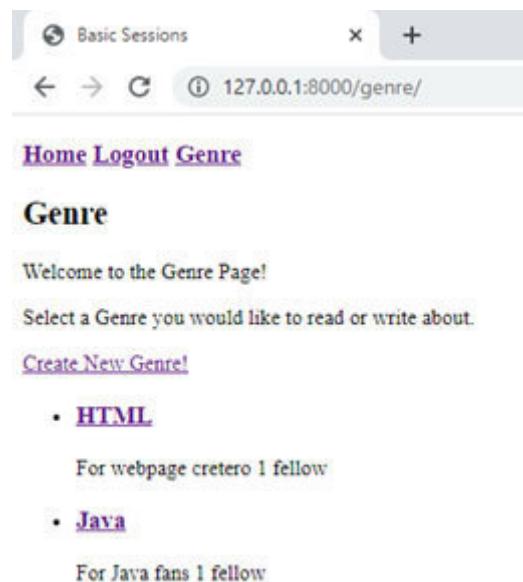


Figure 14.20: Genre page after login

We see the **Create New Genre** link. Click on that:

Basic Sessions

127.0.0.1:8000/genre/new/

Home Logout Genre

Create A New Genre

Name: Python

Description:

If you are a python enthusiast, you are at the right place. Go through our content on how to do stuff using python. Do share your journeys with python.

Create

Figure 14.21: Creating a new genre

Fill in the details and click on

Basic Sessions

127.0.0.1:8000/genre/python/

Home Logout Genre

Python

Fellow Count: 0

[Associate](#)

No posts in this genre yet!

Figure 14.22: Genre detail page

The new **Genre** is created. Click on **Associate** to associate yourself with Python. Once associated, you will start seeing a button to disassociate:

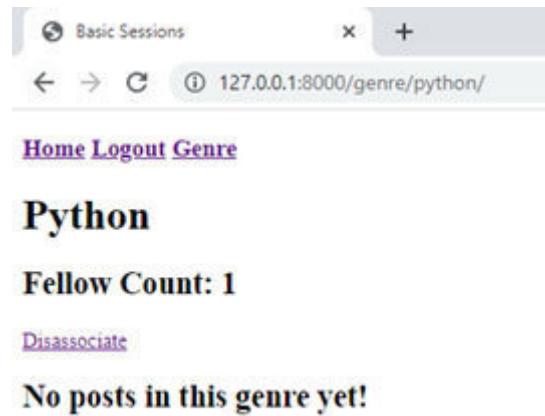


Figure 14.23: Genre detail page after the association

Play along and try to relate the behavior of the site with the code written.

Conclusion

In this chapter, we created the Genre app responsible for categorizing the posts. We worked on the models.py and set up a model that will be used to store the genre information. We created forms that would help us getting data from the user to create new genres and entering them into the database. We created a single genre and list genre views to display the genres existing on the website. We wrote the url-patterns and templates to handle requests and display data on-site.

In the next chapter, we will create the Posts app. This app will hold the content and the genre app will be used to categorize the content. Try and create several Genres related to a field about which you are interested to make a website. I like technology hence, I choose Python, HTML, etc. You can make about movies, literature, books, food, etc. Let's say you choose literature, then your Genres will be like drama, nonfiction, poetry, etc.

Questions

What us the use of slugify?

How can you get the URL to your model objects?

What is the use of

What is the use of the `get_object_or_404` method?

How do you create links to different pages in templates?

CHAPTER 15

The Posts App

Introduction

The most important section of the websites is the posts. This is the reason why users will visit your website – to read and write posts. Posts will have to be linked to the genre app. The relationship with the genre app will help in the segregation of posts and relationships with the account app will help in implementing authorization in the Posts app.

In this chapter, we will not only work on the posts app but also work on some part of the genre app and other project files. This chapter will need more attentive reading to ensure that you do not get confused with the files we are working on.

Structure

Difference between authentication and authorization

Django's built-in authentication and authorization module

Models.py

Views.py

Urls.py

Testing

Conclusion

Objective

The objective of this app is to add handle content of the website. We need to work on the following points:

When a user is not logged in, there is no content on the homepage, but when the user logs in, the content relevant to the user must show up on the homepage. Thus, we will need to update the views.py and templates files of our project such that they display the content relevant to the user on the homepage itself – similar to feeds on any social networking site.

Even if the user is not logged in, we should give the visitors an idea about what our website is so that they find it interesting and might sign up and thus the user size of the website will increase. So, on the genre detail page, we would display all the posts of that genre. To achieve this, we would need to update the views.py file and templates of the genre app.

A model to store details of posts in the database. It should have information like who created the post; the post belongs to which genre, etc.

A view to create the post functionality, a form where users can fill in the details to create a post, and a template to display the form.

A view that displays a single post in detail, a list of all posts, posts by a user, and deletes a post. Templates to display these views.

Register the model in admin.py.

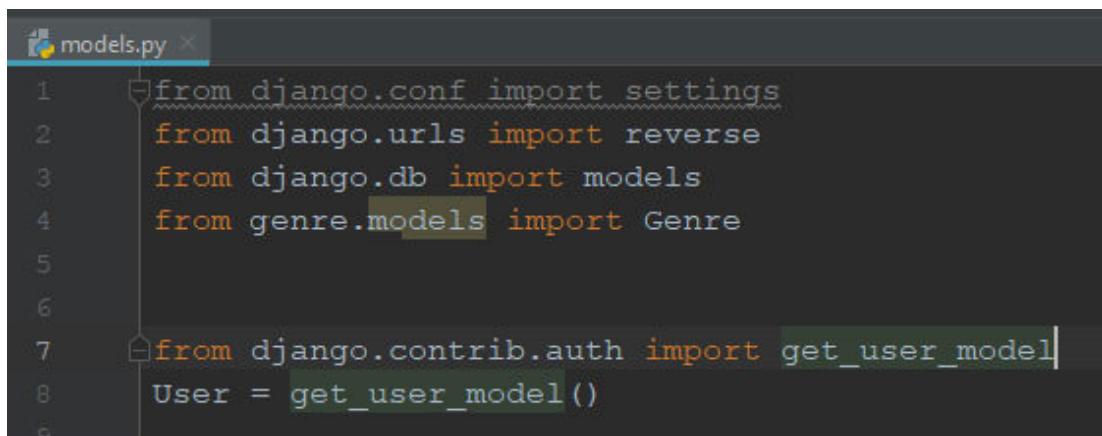
We must create url-patterns for all the views.

Before proceeding, make sure you have created a few users and genres in your website and associated them randomly. This to ensure that now we can create posts; you have users and genres ready.

Let us begin.

models.py

We will need a model to save details of the posts. The details like content, title, author, creation date, etc. will be stored in this model. Go to basicproject/posts/models.py and add the following code to it:



```
models.py
1  from django.conf import settings
2  from django.urls import reverse
3  from django.db import models
4  from genre.models import Genre
5
6
7  from django.contrib.auth import get_user_model
8  User = get_user_model()
```

Figure 15.1: Imports of the models.py file

These are the imports like the models.py of the genre app used for the same purpose. Moving ahead, we will now create the model class Post. In this model, we must have a field like the name of the post, the user who created the post, when was the post created, the content of the post, content_html if the post is to be displayed as HTML, and genre to which the post belongs.

Further, we must have a function which returns the name of the post, a function to save the content values in the content_html field. The get_absoulte_url method is needed o that in the list view

we can easily create links to the individual posts. We also need to declare the sequence in which the post will appear. Like newer posts should be on the top and older posts should be at the bottom.

There must be a constraint that ensures that a genre must have only one post with one name. This is optional but it avoids confusion among the users, so we will add it. Add the following code to your file:

```
11  class Post(models.Model):
12      name = models.CharField(max_length=255, unique=False)
13      user = models.ForeignKey(User, related_name="posts",
14                               on_delete=models.CASCADE)
15      created_at = models.DateTimeField(auto_now=True)
16      content = models.TextField()
17      content_html = models.TextFieldeditable=False)
18      genre = models.ForeignKey(Genre, related_name="posts", null=True,
19                                blank=True, on_delete=models.CASCADE)
20
21
22  @override
23  def __str__(self):
24      return self.name
25
26  @override
27  def save(self, *args, **kwargs):
28      self.content_html = self.content
29      super().save(*args, **kwargs)
30
31  def get_absolute_url(self):
32      return reverse(
33          "posts:single",
34          kwargs={
35              "username": self.user.username,
36              "pk": self.pk
37          }
38
39  class Meta:
40      ordering = ["-created_at"]
41      unique_together = ["user", "name"]
```

Figure 15.2: Post model class

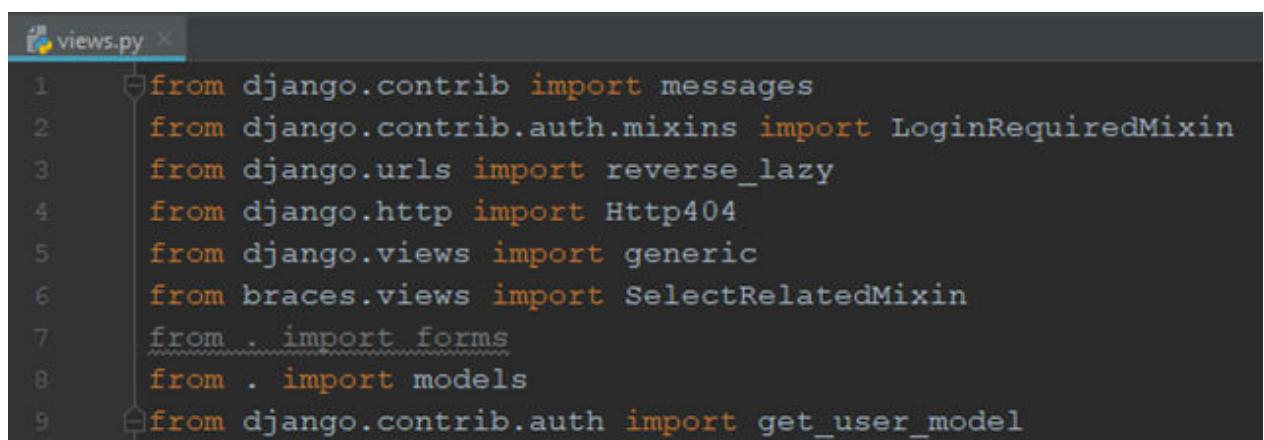
As per the requirement described in the preceding screenshot, this is your model class. In the `get_absolute_url` method, we will use the username and the primary key as a part of the URL. This combination will help in creating a unique URL for each post. When we create the url-pattern for this, we will need to create a pattern such that it accepts the username followed by the primary key Example –

Further, notice the ‘-’ sign before `created_at` in the ordering. This ensures that posts are in descending order of timestamp of creation – newest first.

views.py

Once we have the model (database) ready, we will work on the views (logic to handle the data). Inferring the objectives, we need to work on views.py of the posts app, views.py of basicporoject, and views.py of genre app as well.

Let us begin with basicproject/posts/views.py first. Open the file and add the following code to it:



A screenshot of a code editor showing the contents of a file named 'views.py'. The code contains several import statements. Line 1 imports 'messages' from 'django.contrib'. Line 2 imports 'LoginRequiredMixin' from 'django.contrib.auth.mixins'. Line 3 imports 'reverse_lazy' from 'django.urls'. Line 4 imports 'Http404' from 'django.http'. Line 5 imports 'generic' from 'django.views'. Line 6 imports 'SelectRelatedMixin' from 'braces.views'. Line 7 imports 'forms' from a local directory. Line 8 imports 'models' from a local directory. Line 9 imports 'get_user_model' from 'django.contrib.auth'. The code editor has a dark theme with syntax highlighting.

```
views.py
1  from django.contrib import messages
2  from django.contrib.auth.mixins import LoginRequiredMixin
3  from django.urls import reverse_lazy
4  from django.http import Http404
5  from django.views import generic
6  from braces.views import SelectRelatedMixin
7  from . import forms
8  from . import models
9  from django.contrib.auth import get_user_model
```

Figure 15.3: Imports used in the views.py file

These are imports. Most of them were used previously. The HTTP404 has been imported to display a page not found case scenario in case a request is made to fetch a post of a user that does not exist.

Here, we have an import SelectRelatedMixin in line number 6. For this, you need to install the django-braces library. Inside your

virtualenv after activating it execute the command - pip install django-braces as shown in the following screenshot:

```
(bsenv) C:\Work\basicssessions-root\basicproject>pip install django-braces
```

Figure 15.4: Installing Django-braces

This library provides certain convenient mixins for class-based views which helps in the implementation of authorization. Let us now create views:

PostList view: This view is a generic list view, but it must display the posts which belong to the selected genre or by a selected user. Add the following code to your views.py file:

```
14  class PostList(SelectRelatedMixin, generic.ListView):
15      model = models.Post
16      select_related = ("user", "genre")
```

Figure 15.5: The PostList view class

This view is used to display all the posts by a user. It is a generic List view:

```
19     class UserPosts(generic.ListView):
20         model = models.Post
21         template_name = "posts/user_post_list.html"
22
23         def get_queryset(self):
24             try:
25                 self.post_user = User.objects.prefetch_related("posts").get(
26                     username__iexact=self.kwargs.get("username")
27                 )
28             except User.DoesNotExist:
29                 raise Http404
30             else:
31                 return self.post_user.posts.all()
32
33         def get_context_data(self, **kwargs):
34             context = super().get_context_data(**kwargs)
35             context["post_user"] = self.post_user
36             return context
```

Figure 15.6: The *UserPosts* view class

PostDetail: This view is used to display a post in detail. Add the following code in your file:

```
39     class PostDetail(SelectRelatedMixin, generic.DetailView):
40         model = models.Post
41         select_related = ("user", "genre")
42
43         def get_queryset(self):
44             queryset = super().get_queryset()
45             return queryset.filter(
46                 user__username__iexact=self.kwargs.get("username")
47             )
```

Figure 15.7: The *PostDetail* view class

CreatePost view: This view is used to create a new post. Add the following code to your file:

```
50     class CreatePost(LoginRequiredMixin, SelectRelatedMixin, generic.CreateView):
51         fields = ('name', 'content', 'genre')
52         model = models.Post
53
54     def form_valid(self, form):
55         self.object = form.save(commit=False)
56         self.object.user = self.request.user
57         self.object.save()
58         return super().form_valid(form)
```

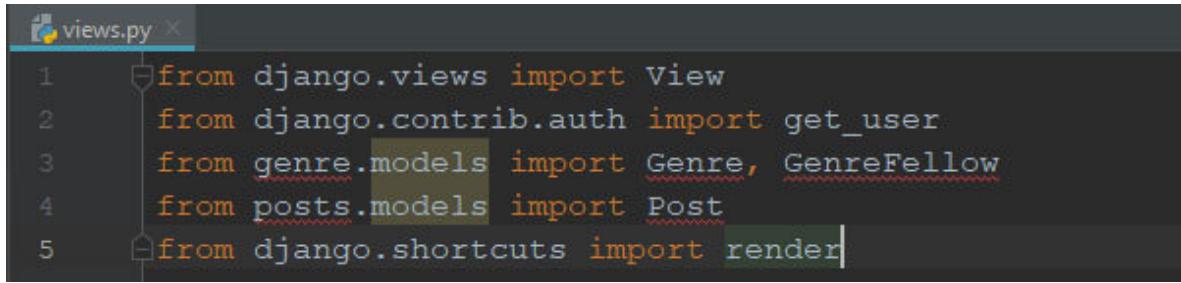
Figure 15.8: The *CreatePost* view class

DeletePost view: This view is used to delete any post. The user who added the post can delete it. Add the following code to your file:

```
61     class DeletePost(LoginRequiredMixin, SelectRelatedMixin, generic.DeleteView):
62         model = models.Post
63         select_related = ("user", "genre")
64         success_url = reverse_lazy("posts:all")
65
66     def get_queryset(self):
67         queryset = super().get_queryset()
68         return queryset.filter(user_id=self.request.user.id)
69
70     def delete(self, *args, **kwargs):
71         messages.success(self.request, "Post Deleted")
72         return super().delete(*args, **kwargs)
```

Figure 15.9: The *DeletePost* view class

These were the views for your posts app. Now, we need to rewrite the view of the project so that we can see posts on the homepage. This is to achieve the first objective mentioned earlier. Open basicproject/basicproject/views.py and delete all existing code and add the following code to it:



```
views.py
1 from django.views import View
2 from django.contrib.auth import get_user
3 from genre.models import Genre, GenreFellow
4 from posts.models import Post
5 from django.shortcuts import render
```

Figure 15.10: Imports for the Project views.py file

The imports are pretty usual. The `get_user` method here is used to fetch the user logged in. If any user is not logged in, this method will return `Anonymous user`.

Now, let's take a look at the Homepage view. Here, the expectation is that if a user is logged in, the HomePage view must show the relevant posts on the homepage itself like the feeds in any social networking site. The criteria for selecting the posts to be displayed may vary. We choose to display the posts which are created by the logged in user and the posts of other users that belong to the genres the logged-in users are associated with.

If any user is not logged in, then there must be no posts displayed on the homepage. Add the following code to your file:

```
8  class HomePage(View):
9
10     def get(self, request, *args, **kwargs):
11         if request.user.is_authenticated:
12             association_list = []
13             post_to_displayed = []
14             user_logged_in = get_user(request)
15             print(user_logged_in.username)
16             for item in GenreFellow.objects.all():
17                 print(item)
18                 if item.user == user_logged_in:
19                     print(item.user)
20                     association_list.append(item.genre)
21             for post_item in Post.objects.all():
22                 if post_item.genre in association_list:
23                     post_to_displayed.append(post_item)
24             return render(request, 'user_home.html',
25                           {'post_list': post_to_displayed})
26         else:
27             return render(request, 'index.html')
```

Figure 15.11: The imported HomePage view class

Here, the Homepage view inherits the class-based view. The get method here will override the get method of the View class. Let's take a look at the get method here line by line.

Firstly, we check whether the user is logged in or not. Then, we create two empty lists. association_list (this will hold the list of genres to which the logged-in user is associated) and post_to_displayed (this is the final list which has all the posts to be displayed and will be rendered to the template).

Then, in line number 16, we iterate through all the objects present in the GenerFellow model. The GenerFellow model is the table that

has records of user and genre associations and appends those genres in the Now, we have the association list.

In line number 21, we iterate through all the posts present and check whether the genre of the post is present in the association list. If yes, we add that post to the post_to_displayed list.

Once we have the post_to_displayed list, we simply render it to the template in line number 24. We will have to create a different template for this part of the view – If the user is not logged in, it will simply render the old template index.html which will work as before.

So, we are done with views.py file of post app and views.py file of basicporoject Now, let us work on the views.py file of the genre app This is to achieve the second objective mentioned earlier.

You must be wondering why we did not complete working on basicproject/genre/views.py in the previous chapter itself. The reason is that we did not have the posts app files then. Now, we have the posts app files, hence we can import them in the basicproject/genre/views.py file and use the Post model there.

Note that out of all the views present in the basicproject/genre/views.py file, we will only edit one view – SingleGenre view and we will add some imports. Rest all the code remains as is.

Firstly, open basicproject/genre/views.py and add the following three imports along with other imports:

```
10     from posts.models import Post
11     from django.views import View
12     from django.shortcuts import render
```

Figure 15.12: Imports to be added to the Genre app views.py file

Here, we imported the Post model from the posts app to the genre app as we need to display the posts on the genre detail template which is triggered by a view present in the genre app views.py file. Now, re-write the SingleGenre class as shown in the following screenshot. Do not edit any other classes:

```
20 class SingleGenre(View):
21
22     def get(self, request, *args, **kwargs):
23         genre_opened = get_object_or_404(Genre, slug=self.kwargs.get("slug"))
24         post_to_displayed = []
25         for item in Post.objects.all():
26             if str(item.genre) == str(genre_opened.name):
27                 post_to_displayed.append(item)
28         return render(request, 'genre/genre_detail.html',
29                         {'post_list': post_to_displayed, 'genre': genre_opened})
```

Figure 15.13: The updated SingleGenre view class

Here we are doing pretty much the same as what we did in the HomePage view class. We fetch the genre opened, and then we iterate through all the posts present in the Post model and pick the post with the genre that is the same as the genre opened. Place all such posts in a list named post_to_displayed and render it to the template. In the last line, you will see that we have rendered two variables to the template. Just to remind, this is how we can render variables to the templates in a dictionary with keys and values. The keys are the variable names and values are their values.

[forms.py](#)

We must create new posts for our posts app. Hence, we need a form for that. Create a basicproject/posts/forms.py file and update it as shown in the following screenshot:

```
from django import forms
from . import models

class PostForm(forms.ModelForm):
    class Meta:
        fields = ('name', 'content', 'genre')
        model = models.Post

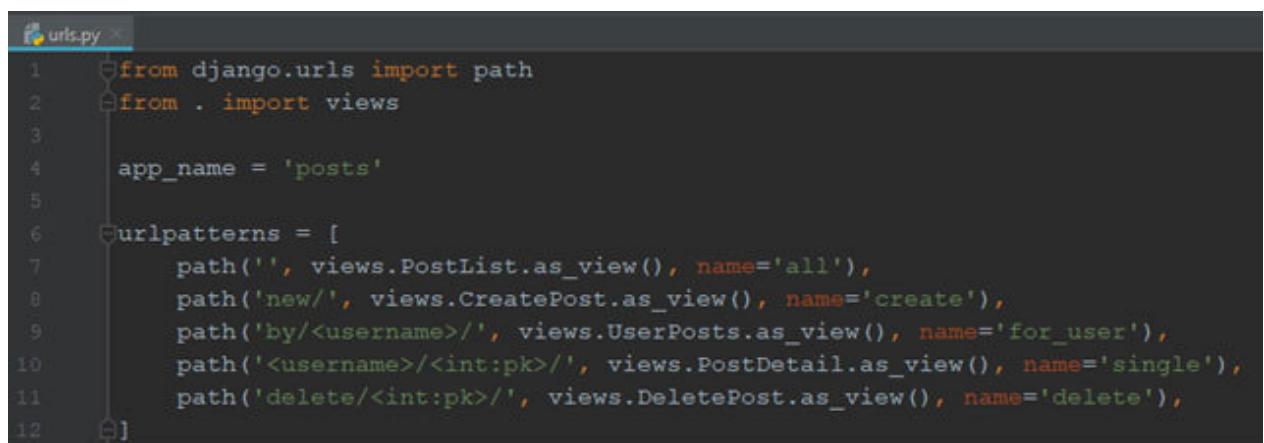
    def __init__(self, *args, **kwargs):
        user = kwargs.pop("user", None)
        super().__init__(*args, **kwargs)
        if user is not None:
            self.fields["genre"].queryset = (
                models.Genre.objects.filter(
                    pk__in=user.genre.values_list("genre_pk")
                )
            )
```

Figure 15.14: The form to create new posts

Here, we are using the name, content, and genre fields of the Post We pick up the user logged in and the details entered by the user in the form and pass it to CreatePost view.

urls.py

Based on the views we have created, we need to write the Open basicproject/posts/urls.py and update it as shown in the following screenshot:



```
urls.py
1  from django.urls import path
2  from . import views
3
4  app_name = 'posts'
5
6  urlpatterns = [
7      path('', views.PostList.as_view(), name='all'),
8      path('new/', views.CreatePost.as_view(), name='create'),
9      path('by/<username>/', views.UserPosts.as_view(), name='for_user'),
10     path('<username>/<int:pk>/', views.PostDetail.as_view(), name='single'),
11     path('delete/<int:pk>/', views.DeletePost.as_view(), name='delete'),
```

Figure 15.15: Url-patterns of post app

Based on the name, let us understand each of these patterns:

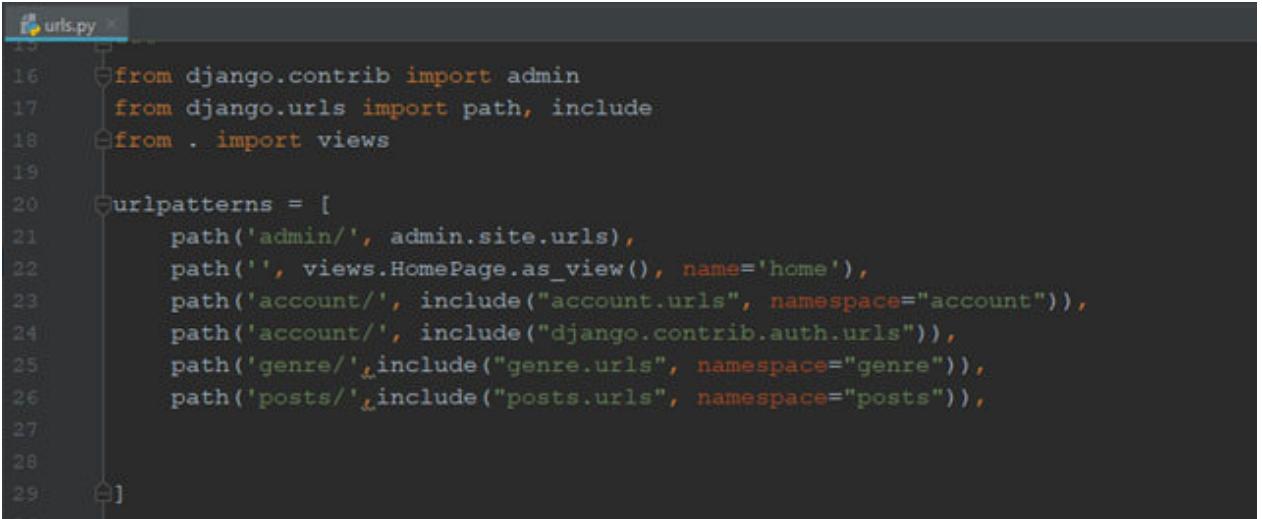
all: It will invoke the post list view - URL formed -

create: It will invoke the CreatePost view - URL formed -

for_user: It will invoke the UserPosts view – URL formed - This URL contains the username of the user logged in. This username is further captured by the view and used to filter out the posts by the user.

delete: It will invoke the DeletePost view – It contains the primary key which is used to fetch the post to be deleted.

single: It invokes the PostDetail view – URL formed - This contains the username of the author and the primary key of the post. This combination is used to filter out the details of the post:



```
urls.py
15
16     from django.contrib import admin
17     from django.urls import path, include
18     from . import views
19
20     urlpatterns = [
21         path('admin/', admin.site.urls),
22         path('', views.HomePage.as_view(), name='home'),
23         path('account/', include("account.urls", namespace="account")),
24         path('account/', include("django.contrib.auth.urls")),
25         path('genre/', include("genre.urls", namespace="genre")),
26         path('posts/', include("posts.urls", namespace="posts")),
27
28
29     ]
```

Figure 15.16: Url-patterns of the project

Also, remember to include the posts url-patterns in the basicproject/basicproject/urls.py file as shown in the preceding screenshot.

Templates

We have a model, views, and url-patterns ready. Now, we need to work on templates. We must create templates for the post app. Also, we must modify templates of the basicproject and genre app to accommodate the first two objectives mentioned earlier.

Let us begin with the templates of the posts app. Go to basicproject/posts/templates/posts/ and create the following files:

_post.html

post_base.html

post_confirm_delete.html

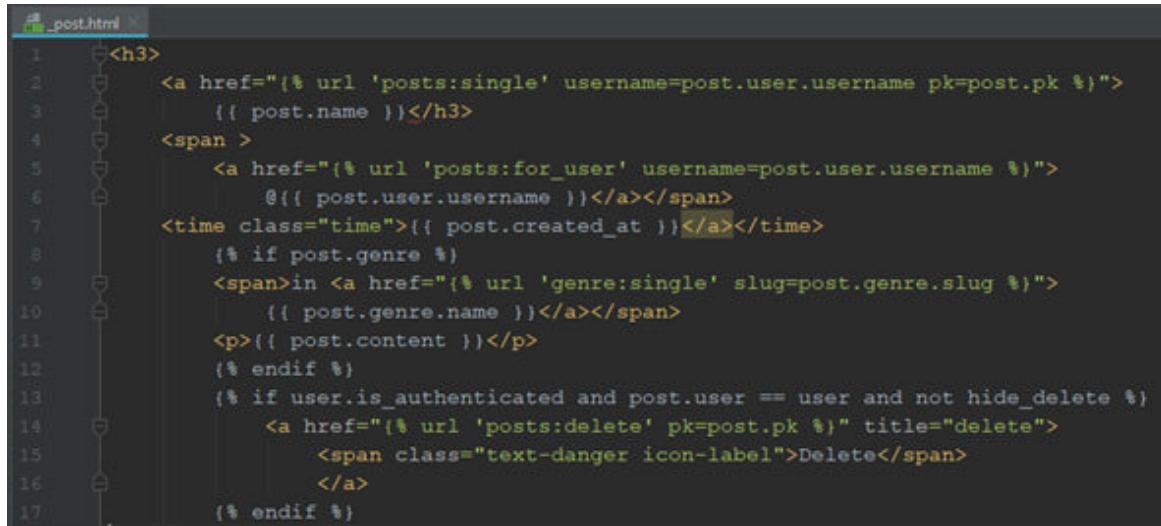
post_detail.html

post_form.html

post_list.html

user_post_list.html

We will work on these templates one by one. Open _post.html and update it as shown in the following screenshot:



The screenshot shows a code editor window with the file '_post.html' open. The code is written in Django's templating language. It includes an H3 tag with a link to a single post, a span containing a link to the user's posts, a timestamp, a genre link, a content paragraph, and a delete link if the user is authenticated and the post belongs to them. The code is numbered from 1 to 17.

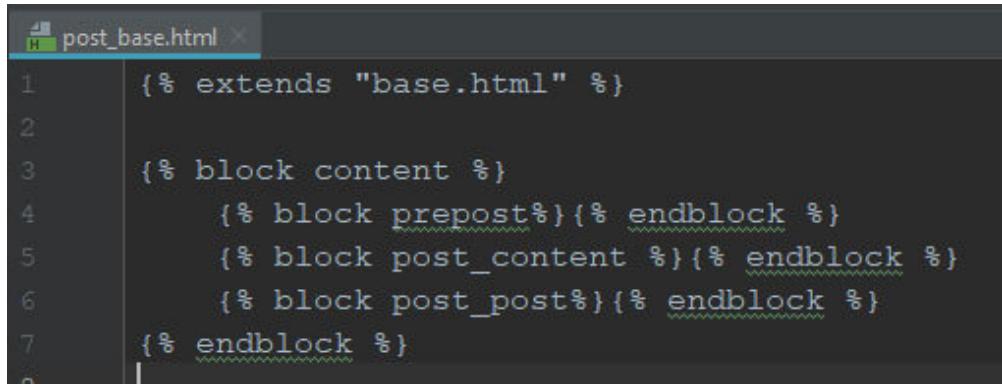
```
1 <h3>
2   <a href="{% url 'posts:single' username=post.user.username pk=post.pk %}">
3     {{ post.name }}</h3>
4   <span>
5     <a href="{% url 'posts:for_user' username=post.user.username %}">
6       @{{ post.user.username }}</a></span>
7     <time class="time">{{ post.created_at }}</a></time>
8     {% if post.genre %}
9       <span>in <a href="{% url 'genre:single' slug=post.genre.slug %}">
10         {{ post.genre.name }}</a></span>
11       <p>{{ post.content }}</p>
12     {% endif %}
13     {% if user.is_authenticated and post.user == user and not hide_delete %}
14       <a href="{% url 'posts:delete' pk=post.pk %}" title="delete">
15         <span class="text-danger icon-label">Delete</span>
16       </a>
17     {% endif %}
```

Figure 15.17: _post.html file

There is a template tag in the Django templating language – include. Suppose you have a certain template code which has to be used in several other templates, you can simply take the common section of the template in a separate file and include that template file in all the other templates where the common code is required.

The _post.html template will be included in several places. This template displays the post name, the username of the creator, timestamp of post creation, and genre to which post belongs, and if the user is authenticated, it gives an option to delete the post. This section is common in several templates; hence, we will include it wherever required.

Open post_base.html and update it as shown in the following screenshot:

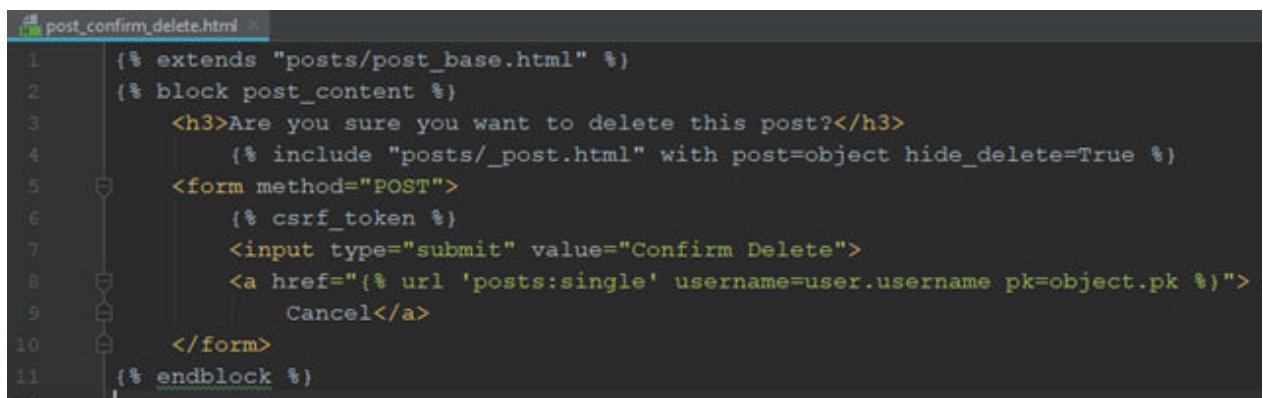


```
post_base.html
1  {% extends "base.html" %}

2
3  {% block content %}
4      {% block prepost%}{% endblock %}
5      {% block post_content %}{% endblock %}
6      {% block post_post%}{% endblock %}
7  {% endblock %}
```

Figure 15.18: post_base.html file

This is just like the genre_base.html you used in the genre app. Open post_confirm_delete.html and update it as shown in the following screenshot:

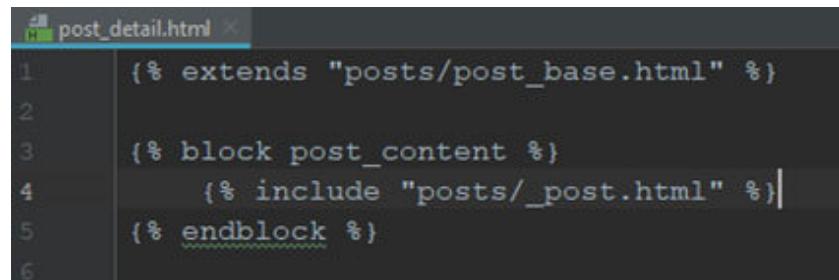


```
post_confirm_delete.html
1  {% extends "posts/post_base.html" %}
2  {% block post_content %}
3      <h3>Are you sure you want to delete this post?</h3>
4      {% include "posts/_post.html" with post=object hide_delete=True %}
5      <form method="POST">
6          {% csrf_token %}
7          <input type="submit" value="Confirm Delete">
8          <a href="{% url 'posts:single' username=user.username pk=object.pk %}">
9              Cancel</a>
10         </form>
11     {% endblock %}
```

Figure 15.19: post_confirm_delete.html file

The DeleteView which we inherited our DeletePost view takes a template with suffix _confirm_delete.html by default for confirmation of the deletion. The standard practice is that we put the model name as prefix and create a template for this purpose. The model name is posted; hence, It is rendered by the generic view itself.

Open post_detail.html and update it as shown in the following screenshot:



```
post_detail.html
1  {% extends "posts/post_base.html" %} 
2
3  {% block post_content %} 
4      {% include "posts/_post.html" %} 
5  {% endblock %}
```

Figure 15.20: post_base.html file

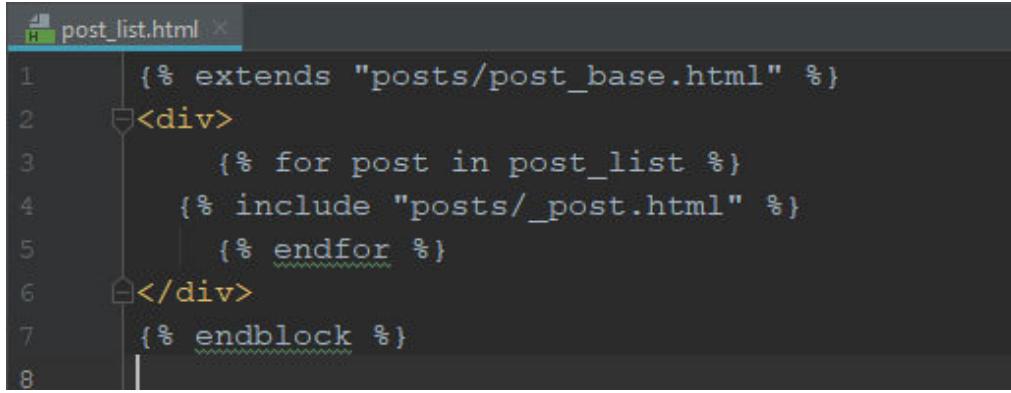
This is your post detail template. This is how we use the include tag to include HTMLs from other template files. It will display whatever the _post.html has. Open post_form.html and update it as shown in the following screenshot:



```
post_form.html
1  {% extends "posts/post_base.html" %} 
2  {% block post_content %} 
3      <h4>Create New Post</h4> 
4      <form method="POST" action="{% url 'posts:create' %}" id="postForm"> 
5          {% csrf_token %} 
6          {{ form }} 
7          <input type="submit" value="Post"> 
8      </form> 
9  {% endblock %}
```

Figure 15.21: post_form.html file

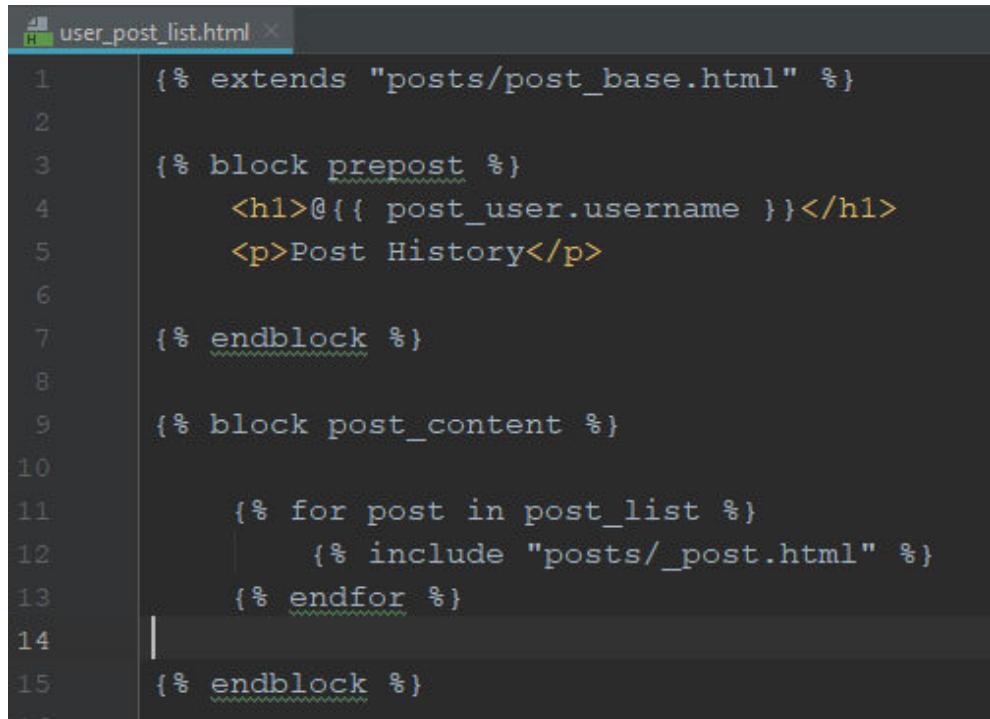
This post_form.html template is used to render the form for creating new posts. Open the post_list.html file and update it as shown in the following screenshot:



```
post_list.html
1  |  {% extends "posts/post_base.html" %} 
2  |  <div>
3  |      {% for post in post_list %}
4  |          {% include "posts/_post.html" %}
5  |      {% endfor %}
6  |  </div>
7  |  {% endblock %}
```

Figure 15.22: *post_list.html* file

Again, we have used The post-filtering is handled in the view. Open the post_user_list.html file. Update it as shown in the following screenshot:



```
user_post_list.html
1  |  {% extends "posts/post_base.html" %} 
2
3  |  {% block prepost %}
4  |      <h1>@{{ post_user.username }}</h1>
5  |      <p>Post History</p>
6
7  |  {% endblock %}
8
9  |  {% block post_content %}
10
11 |      {% for post in post_list %}
12 |          {% include "posts/_post.html" %}
13 |      {% endfor %}
14
15 |  {% endblock %}
```

Figure 15.23: *user_post_list.html*

This template is used to display all the posts by any user. The post-filtering is handled in the UserPost view. Again, we have included the _post.html file. These were the templates of the post app. Now, let us work on the templates of the basicproject to achieve objective 1.

In the create a user_home.html file. This file is in the same folder as the index.html file. Update the basicproject/templates/user_home.html file as shown in the following screenshot:



The screenshot shows a code editor window with the title "user_home.html". The code is written in Python's Jinja2 templating language. It starts with an extends tag to inherit from "base.html". Inside the content block, it displays a welcome message and then checks if the user is authenticated. If true, it prints a greeting and loops through a list of posts, rendering each one using the "_post.html" template. If the user is not authenticated, it simply ends the block. The code is numbered from 1 to 10 on the left.

```
1  {% extends "base.html" %}\n2  {% block content %}\n3      <h1> Welcome to Basic Sessions</h1>\n4      {% if user.is_authenticated %}\n5          <p>Hello {{ user.username }}. </p>\n6          {% for post in post_list %}\n7              {% include "posts/_post.html" %}\n8          {% endfor %}\n9      {% endif %}\n10     {% endblock %}
```

Figure 15.24: user_home.html file

This template also includes the _post.html file. It checks whether the user is logged in and if yes, it displays the posts available from the HomePage view in the basicproject/basicproject/views.py file. If you check the HomePage view, it renders the user_home.html if the user is logged in and index.html if the user is not logged in. This solves the objective to show relevant posts to the user on the homepage.

Now, let's create a template where unauthenticated users can get the look and feel of your site. We will edit the genre_detail.html template. If you take a look at the basicproject/genre/templates/genre/genre_detail.html file, you will see that it has a top block and a middle block. The middle block has the text No posts in this genre yet! Just delete the middle block from block middle to endblock and write the following code instead:

```
16      {% block middle %} 
17          {% for post in post_list %} 
18              <h3><a href="{% url 'posts:single' username=post.user.username pk=post.pk %}"> 
19                  {{ post.name }}</h3> 
20              <span><a href="{% url 'posts:for_user' username=post.user.username %}"> 
21                  @{{ post.user.username }}</a></span> 
22              <time class="time">{{ post.created_at }}</a></time> 
23                  <p>{{ post.content }}</p> 
24                  {% if user.is_authenticated and post.user == user and not hide_delete %} 
25                      <a href="{% url 'posts:delete' pk=post.pk %}" title="delete"> 
26                          <span class="text-danger icon-label">Delete</span> 
27                      </a> 
28                  {% endif %} 
29          {% empty %} 
30      {% endfor %} 
31  {% endblock %}
```

Figure 15.25: Updated middle block of the genre_detail.html file

This template displays the posts of any genre in the post detail page. This will help the unauthenticated users to understand the type of content present on the website.

Congratulations! That is all the coding needed from the Django perspective for this project to run successfully on your local. You can enhance it as you wish. You can use features such as user verification, emailing, etc. You can work on the frontend and make it look beautiful and add effects.

Let us test this now.

Testing

We will go through the entire journey like logging in, creating a post, viewing the posts, deleting a post, logging out, view the genres, and post without logging in. Let us begin. Execute the following command in your terminal.

```
python manage.py makemigrations genre  
python manage.py migrate  
python manage.py runserver
```

If you have not made any errors, you will be able to see the following screen:

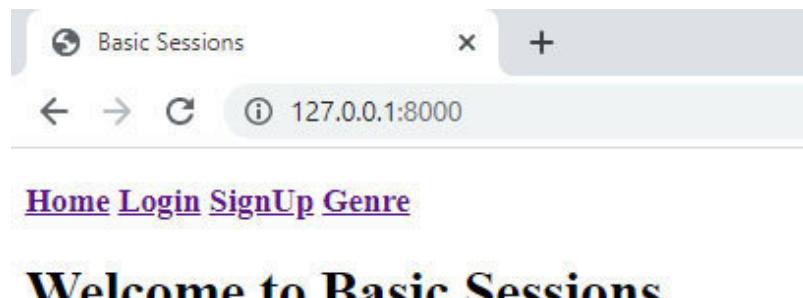


Figure 15.26: Home page without login

The homepage of the site is displayed in the following screenshot:

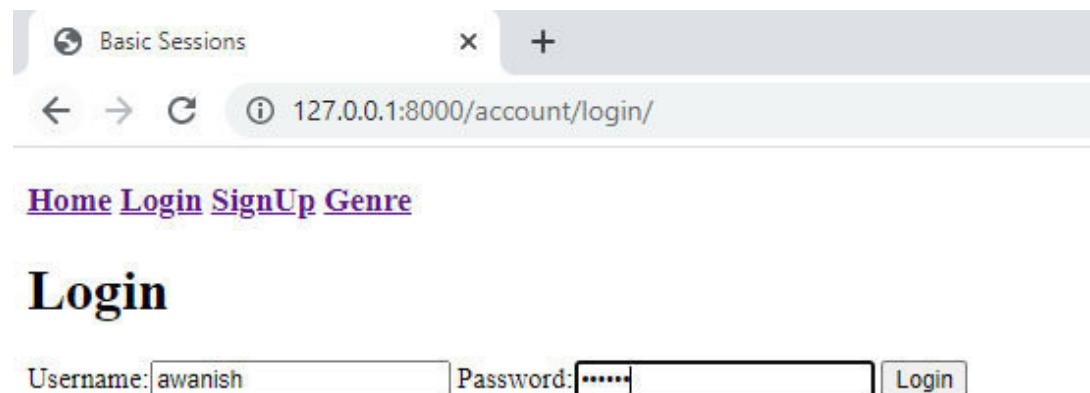


Figure 15.27: Login page

The login screen is displayed in the following screenshot:

A screenshot of a web browser window titled "Basic Sessions". The address bar shows the URL "127.0.0.1:8000". Below the address bar, there are navigation links: Home, Genres, Write a new Post, and Logout. The main content area has a title "Welcome to Basic Sessions". It displays a message "Hello awanish." followed by a list of posts:

- HTML and CSS**
awanish May 30, 2020, 1:09 p.m. in [HTML](#)
HTML and css go hand in hand
[Delete](#)
- Web Scraping with Python**
awanish May 29, 2020, 3:29 p.m. in [Python](#)
Imagine you have to pull a large amount of data from websites and you want to do it as quickly as possible. How would you do it without manually going to each Web Scraping just makes this job easier and faster.
[Delete](#)
- HTML5**
awanish May 28, 2020, 5:33 p.m. in [HTML](#)
HTML5 is a markup language used for structuring and presenting content on the World Wide Web. It was the fifth and last major version of HTML that is a World
[Delete](#)
- Python scripting**
awanish May 28, 2020, 5:29 p.m. in [Python](#)
Featured snippet from the web The interpreter reads and executes each line of code one at a time, just like a SCRIPT for a play or an audition, hence the the term '

Figure 15.28: The homepage after login

The homepage after the user has logged in is displayed in the followed screenshot. Now, we can see the posts relevant to the user on the homepage:



[Home](#) [Genres](#) [Write a new Post](#) [Logout](#)

Python

Fellow Count: 2

[Disassociate](#)

[Web Scraping with Python](#)

[@awanish](#) May 29, 2020, 3:29 p.m.

Imagine you have to pull a large amount of data from websites and you want to do it as quickly as possible. Scraping just makes this job easier and faster.

[Delete](#)

[Python scripting](#)

[@awanish](#) May 28, 2020, 5:29 p.m.

Featured snippet from the web The interpreter reads and executes each line of code one at a time. It's like running a program line by line. That's why it's called a scripting language.

[Delete](#)

Figure 15.29: Genre detail page

The genre detail page shows the posts in the genre:

Basic Sessions

127.0.0.1:8000/posts/new/

[Home](#) [Genres](#) [Write a new Post](#) [Logout](#)

Create New Post

Name:

Content:

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source. Ridiculously fast.

Genre:

Figure 15.30: Creating a new post

Creating a new post is shown in the following screenshot:

Basic Sessions

127.0.0.1:8000/posts/by/awanish/10/

[Home](#) [Genres](#) [Write a new Post](#) [Logout](#)

[Python - Django](#)

@awanish May 30, 2020, 4:57 p.m. in [Python](#)

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced dev... your app without needing to reinvent the wheel. It's free and open source. Ridiculously fast.

[Delete](#)

Figure 15.31: Post detail page

The post detail page will look like the following screenshot:



[Home](#) [Login](#) [SignUp](#) [Genre](#)

Python

Fellow Count: 2

Login to associate to this genre.

[Python - Django](#)

[@awanish](#) May 30, 2020, 4:57 p.m.

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. It's built by experienced Python developers and designed to make your app without needing to reinvent the wheel. It's free and open source. Ridiculously fast.

[Web Scraping with Python](#)

[@awanish](#) May 29, 2020, 3:29 p.m.

Imagine you have to pull a large amount of data from websites and you want to do it as quickly as possible. Scraping just makes this job easier and faster.

[Python scripting](#)

[@awanish](#) May 28, 2020, 5:29 p.m.

Featured snippet from the web The interpreter reads and executes each line of code one at a time, just like a computer program. That's why it's called a scripting language.

Figure 15.32: Genre detail page without login

In the preceding image you can see the genre detail page for an unauthenticated user. The post is showing up.

Conclusion

In this chapter, we created the Post app responsible for creating the posts. We worked on the models.py and set up a model that will be used to store post information. We created forms that would help us getting data from the user to create a new post and entering them into the database. We created a post, list post, etc. views to display the posts existing on the website. We wrote the url-patterns and templates to handle requests and display data on-site.

The theory chapters will give you an understanding of what Django provides and from which imports you can get them. It will help you browse through the documentation. The project chapter will give you an understanding of how to use tools like classes, methods, functions, etc. provided by Django. Once you get to know how to find the relevant Django tool and how to use it, you will be able to code any feature into your project.

In the next chapter, we will learn how to deploy the website. We will use Heroku to deploy our Django site. Heroku is the most favored tool to deploy Django projects for learning and practicing purposes. You can use it for production sites but that are paid and performance is limited.

Questions

How do create a foreign key relationship between two models?

How do you set the ordering of the objects of any model?

How to get context data from any model?

How do you check whether a user is authenticated?

How to reuse a template in another template?

CHAPTER 16

Deploying the Website

Introduction

Your project is ready and tested. If you wish to share it with the world, you need to put it on a publicly accessible URL. The current server is your local and only you can access it. Deploy it on a server and connect it to a public URL.

You now have a clear picture of the Django code in the files. Now, you may choose to enhance your frontend or deploy it as is. In this chapter, we will deploy the website we created in the previous chapter. We will use the free service on Heroku. Heroku will pull your code to a repository and provide a server on which your project will run.

Before we deploy your project, read it first and understand the process. It will be much easier if you follow the chapter to deploy your project after reading it once. You will save a lot of rework.

Structure

Deployment configurations in the settings.py file

Setting up Heroku

How to get a custom domain name

Conclusion

Objective

To deploy the website on a domain that can be accessed publicly.

Deployment configurations in the project

Before we go ahead and move our code to GitHub, we need to write some configuration code in the project. Open basicproject/basicproject/settings.py and add the following line of code in it. Along with the STATIC_URL and add another variable STATIC_ROOT as shown in the following screenshot:

```
STATIC_URL = '/static/'  
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]  
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Figure 16.1: STATIC_ROOT variable

In the MIDDLEWARE list, add a middleware whitenoise.middleware.WhiteNoiseMiddleware as shown in the following screenshot:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'whitenoise.middleware.WhiteNoiseMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

Figure 16.2: Adding whitenoise middleware

These changes were done so that Heroku can run and collect static commands and keep all the static files in the root folder. From the root folder, all the static content can be rendered. Now, our site needs to be hosted by an external server so we need to add a host in the allowed host. Think of a unique name for your app. I will use Update the allowed host list as shown in the following screenshot:

```
ALLOWED_HOSTS = ['basicssessions.herokuapp.com', '127.0.0.1']
```

Figure 16.3: Adding domains to ALLOWED_HOSTS variable

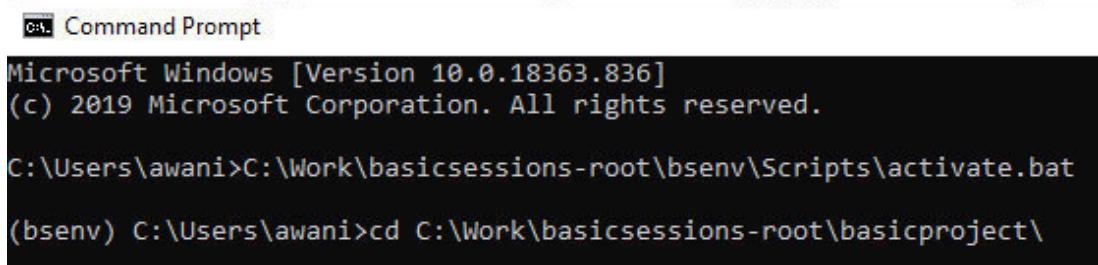
The basicsession.herokuapp.com will be our public website or web address. We have added 127.0.0.1 so that we do not have to make changes to this file every time we run this project locally. The project will now run on these two hosts only. If want to run it on any other host, you will have to add it here.

Note that basicsession is the name that will be given to the Heroku app on the Heroku server. If the name you choose is already being used by someone else, you won't be able to use that name. Hence, it should be unique. With a public domain, your site can be visited by anyone who has a web address. Thus, you need to ensure that the internal code is displayed on the public site if any error occurs on your site. Set the Debug = False from

```
DEBUG = False
```

Figure 16.4: Setting DEGUB = False

Now, go to the folder where your manage.py file is present. Here, you have to create three files. For this, open Command Prompt and change directory to the project folder which contains the manage.py file:



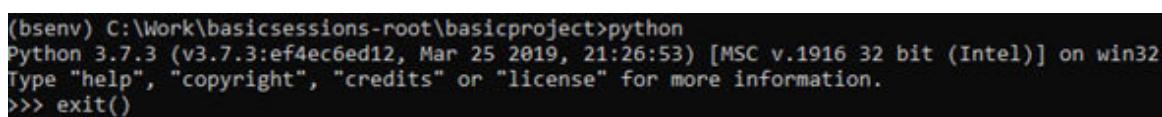
```
Microsoft Windows [Version 10.0.18363.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\awani>C:\Work\basicsessions-root\bsenv\Scripts\activate.bat

(bsenv) C:\Users\awani>cd C:\Work\basicsessions-root\basicproject\
```

Figure 16.5: Activating virtual environment.

Create a text file and name it In this file, write the version of Python you have in your project. To find out the Python version, open Command Prompt and activate the virtualenv of your project as shown in the following screenshot:



```
(bsenv) C:\Work\basicsessions-root\basicproject>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Figure 16.6: Checking Python version

This file will be used by Heroku to fetch the Python version to be installed on the server. Your runtime.txt file should look like the following screenshot:

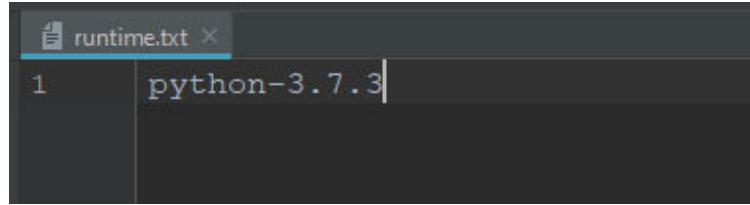


Figure 16.7: The runtime.txt file

Now, create a file named Note that the file should have no extensions. To create this file in the Command Prompt change directory in the folder with manage.py file and execute the command – notepad

```
(bsenv) C:\Work\basicsessions-root\basicproject>dir
Volume in drive C is Windows
Volume Serial Number is 4883-E662

Directory of C:\Work\basicsessions-root\basicproject

31-05-2020  13:51    <DIR>      .
31-05-2020  13:51    <DIR>      ..
30-05-2020  22:51    <DIR>      account
31-05-2020  14:05    <DIR>      basicproject
31-05-2020  13:36            188,416 db.sqlite3
30-05-2020  22:51    <DIR>      genre
24-05-2020  01:11            653 manage.py
30-05-2020  22:51    <DIR>      posts
31-05-2020  13:53            45 Procfile
31-05-2020  13:37            135 requirements.txt
31-05-2020  13:51            12 runtime.txt
30-05-2020  22:51    <DIR>      static
30-05-2020  22:51    <DIR>      templates
      5 File(s)        189,261 bytes
      8 Dir(s)  109,740,023,808 bytes free

(bsenv) C:\Work\basicsessions-root\basicproject>notepad Procfile
```

Figure 16.8: Command to create Procfile

Then, the following screen will pop up:

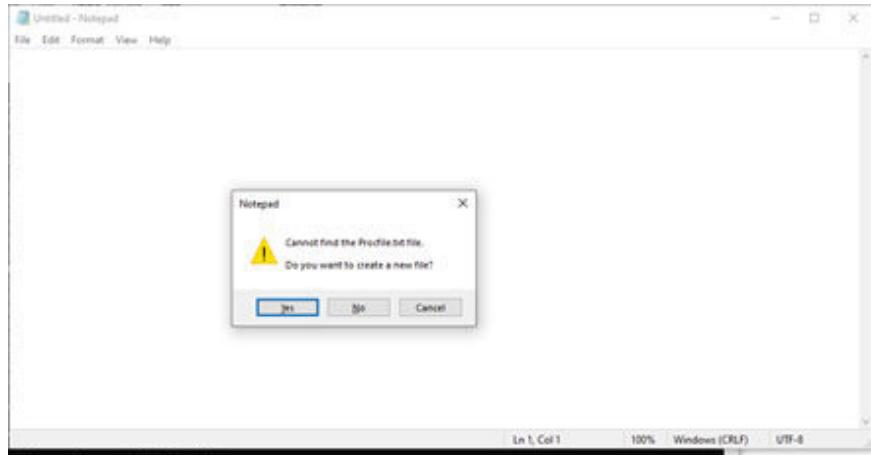


Figure 16.9: Response of the command in [Figure 16.8](#)

Click on yes and write the text as shown in the following screenshot and save the file:

```
*Procfile - Notepad
File Edit Format View Help
web: gunicorn basicproject.wsgi --log-file -
```

A screenshot of a Windows Notepad window titled "*Procfile - Notepad". The menu bar includes "File", "Edit", "Format", "View", and "Help". The main text area contains the command "web: gunicorn basicproject.wsgi --log-file -". The Notepad window has a standard Windows title bar and status bar at the bottom.

Figure 16.10: The Procfile

This is the file Heroku will refer to find the wsgi file for the project.

We have used a middleware and whitenoise and a new module gunicorn. We need to install these in your project. Execute the following command with your virtualenv active as shown in the following screenshot:

```
pip install gunicorn whitenoise
```

```
(bsenv) C:\Work\basicsessions-root\basicproject>pip install gunicorn whitenoise
```

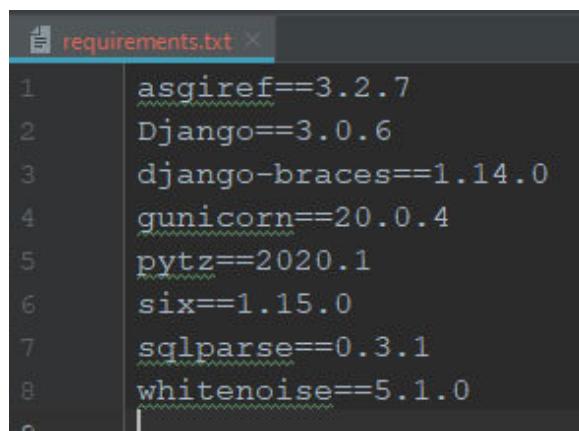
Figure 16.11: Command to install gunicorn and whitenoise

Now we need to tell Heroku to install all the requirements that are needed by our project in the server. Execute the command pip freeze >

```
(bsenv) C:\Work\basicsessions-root\basicproject>pip freeze > requirements.txt  
(bsenv) C:\Work\basicsessions-root\basicproject>
```

Figure 16.12: Command to list out dependencies of your project

This will create a list of installations used in your project currently:



A screenshot of a code editor window titled "requirements.txt". The file contains the following list of dependencies:

```
1 asgiref==3.2.7
2 Django==3.0.6
3 django-braces==1.14.0
4 gunicorn==20.0.4
5 pytz==2020.1
6 six==1.15.0
7 sqlparse==0.3.1
8 whitenoise==5.1.0
```

Figure 16.13: The requirements.txt file showing the dependencies of the project

This is how your requirements.txt file should look like. We have our project ready to be deployed.

Setting up Heroku

Heroku is a hosting platform. Here, you can use the web address of Heroku and host your websites for free. You will have to pay if you want a custom domain. For example, Hosting with the web address www.basicsession.herokuapp.com is free, but if I want to host with the web address I will have to pay for it.

We will go with the free option. Open <https://www.heroku.com/> and sign up if you do not have an account. Fill in the details on the following page:

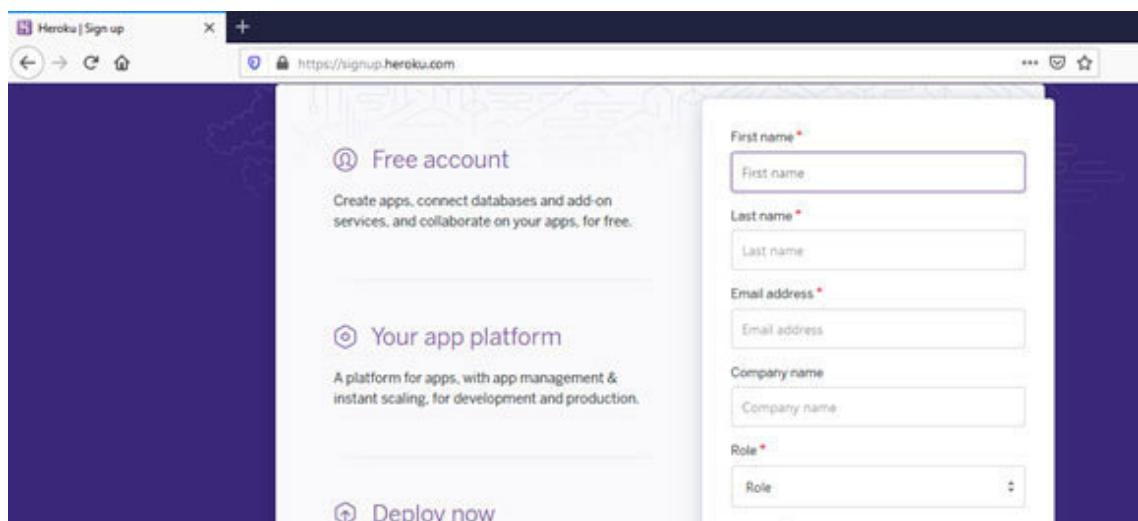


Figure 16.14: The Heroku Signup page

Once you have filled the details, click on **Create Free**. You will have to confirm your email. Log in to the email you provided and confirm your Heroku account. You will be prompted to create a password.

Once done, log in to your Heroku account, and you will see your dashboard:

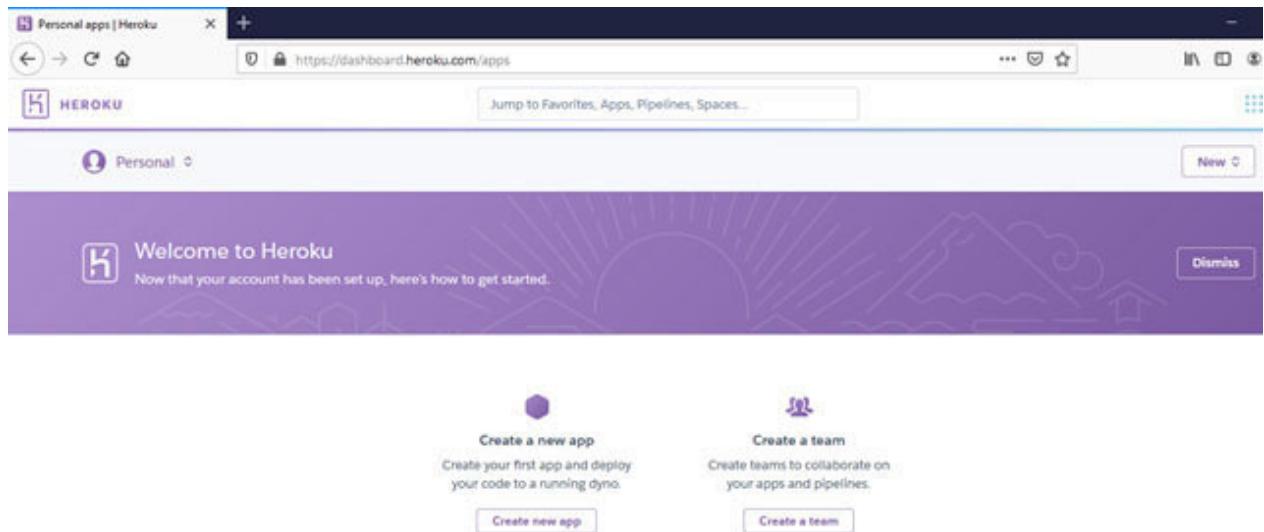


Figure 16.15: Heroku user dashboard

Click on create a new app. This is where you need to put the unique app name you put in the allowed hosts of the setting.py file. If the app name you have set in the allowed host is not available on Heroku, then select any other app name. Then, go to your settings.py file and update the app name in the allowed hosts in the settings.py file:

App name

 ✓

basicsession is available

Choose a region

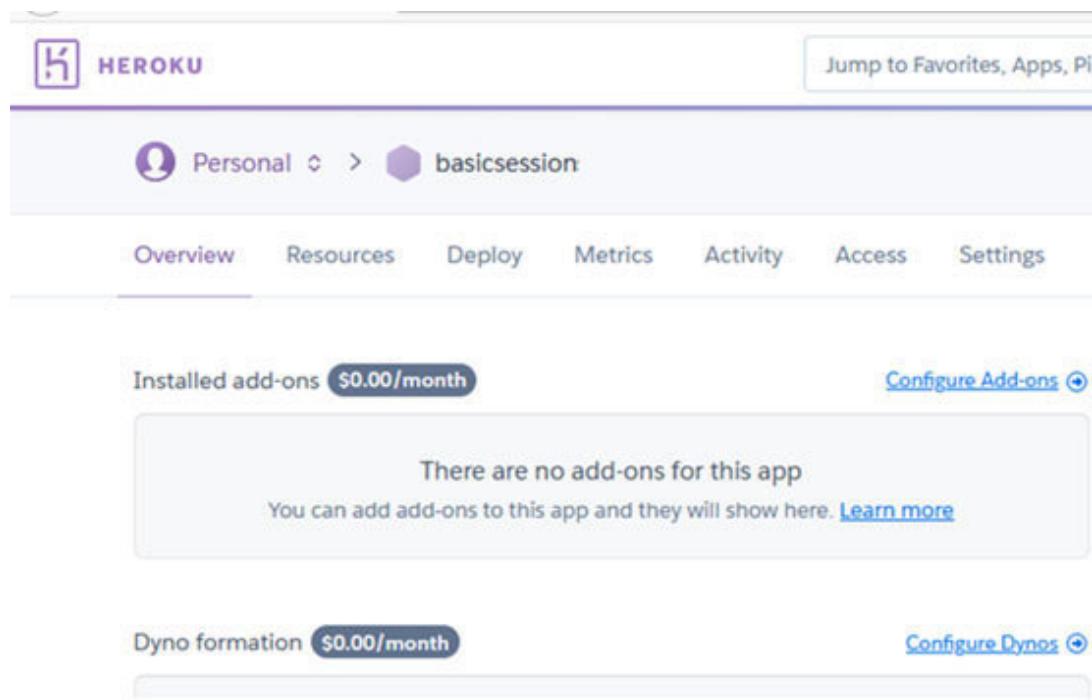
 ▼

 Add to pipeline...

Create app

Figure 16.16: Creating a new app in Heroku

Click on the create app. Your app will be created and you will be taken to the app dashboard:



The screenshot shows the Heroku app dashboard for the application 'basicsession'. At the top, there's a navigation bar with the Heroku logo, a 'Personal' dropdown, and a 'Jump to Favorites, Apps, Pi' link. Below the navigation, the app name 'basicsession' is displayed next to a purple hexagon icon. A horizontal menu bar includes 'Overview', 'Resources', 'Deploy', 'Metrics', 'Activity', 'Access', and 'Settings'. The 'Overview' tab is selected. In the main content area, there's a section for 'Installed add-ons' showing '\$0.00/month' and a 'Configure Add-ons' button. A message box states 'There are no add-ons for this app' and provides a link to 'Learn more'. Another section for 'Dyno formation' shows '\$0.00/month' and a 'Configure Dynos' button.

Figure 16.17: Heroku app dashboard

Now, click on the **Deploy** tab. Here, you will see three different methods of deploying. Go to the **Deployment Method** and click on **Heroku**

The screenshot shows the 'Deployment Method' section of the Heroku deployment interface. It features three main options:

- Heroku Git**: Represented by a purple icon and the text "Use Heroku CLI".
- Github**: Represented by a GitHub logo and the text "Connect to GitHub".
- Container Registry**: Represented by a blue hexagonal icon and the text "Use Heroku CLI".

Install the Heroku CLI

[Download and Install the Heroku CLI](#)

If you haven't already, log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

Create a new Git repository

[Initialize a git repository in a new or existing directory](#)

```
$ cd my-project/
$ git init
$ heroku git:remote -a basicsession
```

Deploy your application

[Commit your code to the repository and deploy it to Heroku using Git.](#)

```
$ git add .
$ git commit -am "make it better"
$ git push heroku master
```

Figure 16.18: Step to deploy via Heroku CLI

We will follow the steps mentioned in the preceding screenshot. Let us begin by installing the Heroku CLI. Open

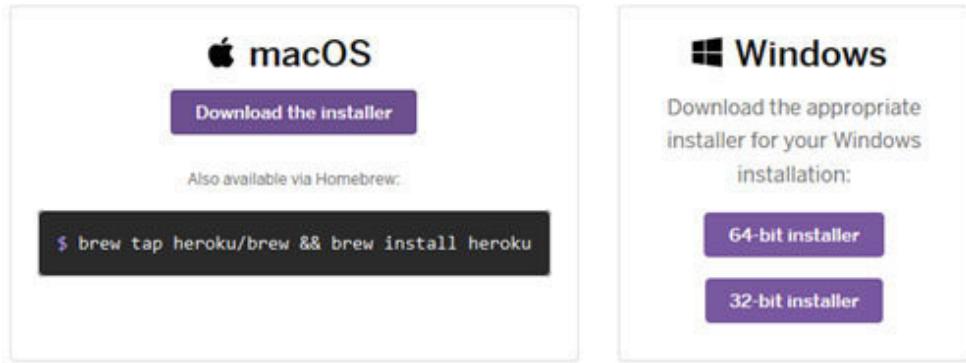


Figure 16.19: Heroku CLI download links

Click on the version relevant to your system. The setup will be downloaded. Download and install the Heroku CLI. Then, open a new Command Prompt and follow the given commands:

heroku login

```
Windows PowerShell
Microsoft Windows [Version 10.0.18363.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\awani>heroku login
heroku: Press any key to open up the browser to login or q to exit:
```

Figure 16.20: Logging in to Heroku using CLI

A browser will open where you will have to log in to your Heroku account. Once done, the Heroku account will be logged in Command Prompt to your project folder (the folder where the manage.py file is present)):

git init

heroku git:remote -a basicsession

```
C:\Users\awani>cd C:\Work\basicsessions-root\basicproject  
C:\Work\basicsessions-root\basicproject>git init  
Initialized empty Git repository in C:/Work/basicsessions-root/basicproject/.git/  
C:\Work\basicsessions-root\basicproject>heroku git:remote -a basicsession  
set git remote heroku to https://git.heroku.com/basicsession.git
```

Figure 16.21: Initiating a Git repo on Heroku git

Your Heroku git repository will be setup:

git add.

```
C:\Work\basicsessions-root\basicproject>git add .  
warning: LF will be replaced by CRLF in account/templates/account/login.html.  
The file will have its original line endings in your working directory  
warning: LF will be replaced by CRLF in account/templates/account/signup.html.  
The file will have its original line endings in your working directory  
warning: LF will be replaced by CRLF in genre/templates/genre/genre_base.html.  
The file will have its original line endings in your working directory
```

Figure 16.22: Add all the files of the project folder to staging for commit

All the files in your project directory will be staged for commit:

git commit -am "make it better"

```
C:\Work\basicsessions-root\basicproject>git commit -am "make it better"  
[master (root-commit) e2fc87a] make it better  
 94 files changed, 1021 insertions(+)  
   create mode 100644 Procfile  
   create mode 100644 Procfile.txt  
   create mode 100644 account/__init__.py  
   create mode 100644 account/_pycache_/_init__.cpython-37.pyc  
   create mode 100644 account/_pycache_/_admin.cpython-37.pyc
```

Figure 16.23: Committing the staged files to the Heroku git repo

All the files are committed:

```
git push heroku origin
```

```
C:\Work\basicssessions-root\basicproject>git push heroku master
Enumerating objects: 111, done.
Counting objects: 100% (111/111), done.
Delta compression using up to 8 threads
Compressing objects: 100% (101/101), done.
Writing objects: 100% (111/111), 42.32 KiB | 1.46 MiB/s, done.
```

Figure 16.24: Pushing the committed files to the Heroku repo

The code from your local machine is pushed to the Heroku repository. Once the code is pushed, deployment starts automatically. You can see the logs in the Command Prompt:

```
Total 111 (delta 17), reused 0 (delta 0), pack-reused 0
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: !     Python has released a security update! Please consider upgrading
remote:     Learn More: https://devcenter.heroku.com/articles/python-runtime-environment
remote: -----> Installing python-3.7.3
remote: -----> Installing pip
remote: -----> Installing SQLite3
remote: -----> Installing requirements with pip
remote:     Collecting asgiref==3.2.7
remote:         Downloading asgiref-3.2.7-py2.py3-none-any.whl (19 kB)
remote:     Collecting Django==3.0.6
remote:         Downloading Django-3.0.6-py3-none-any.whl (7.5 MB)
remote:     Collecting django-braces==1.14.0
remote:         Downloading django_braces-1.14.0-py2.py3-none-any.whl (14 kB)
remote:     Collecting gunicorn==20.0.4
remote:         Downloading gunicorn-20.0.4-py2.py3-none-any.whl (77 kB)
remote:     Collecting pytz==2020.1
remote:         Downloading pytz-2020.1-py2.py3-none-any.whl (510 kB)
remote:     Collecting six==1.15.0
remote:         Downloading six-1.15.0-py2.py3-none-any.whl (10 kB)
remote:     Collecting sqlparse==0.3.1
remote:         Downloading sqlparse-0.3.1-py2.py3-none-any.whl (40 kB)
remote:     Collecting whitenoise==5.1.0
remote:         Downloading whitenoise-5.1.0-py2.py3-none-any.whl (19 kB)
remote:     Installing collected packages: asgiref, pytz, sqlparse, Django
remote:     Successfully installed Django-3.0.6 asgiref-3.2.7 django-braces-1.14.0
remote: -----> $ python manage.py collectstatic --noinput
remote:     131 static files copied to '/tmp/build_b3b3b0eaaf08e32d0ccd41e1'
remote:
remote: -----> Discovering process types
remote:     Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:     Done: 54M
remote: -----> Launching...
remote:     Released v5
remote:     https://basicsession.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
```

Figure 16.25: Automatic deployment starts and the website URL is displayed

The deployment is complete. You can see the URL of your website. Open that URL:

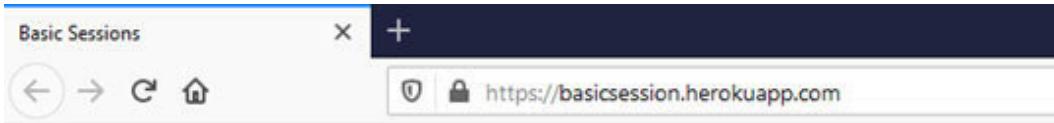


Figure 16.26: Opening the public URL of the project on browser

Bingo! There is your website.

Test it. Share it -

This was a quick deployment. We used the Heroku hosting service. You can enhance it by working on the frontend. After you have made changes to your code, you can deploy it to the same site. You will just need to follow the steps.

Commit your code to the repository and deploy it to Heroku using Git:

```
$ git add.  
$ git commit -am "make it better".  
$ git push Heroku master.
```

Conclusion

We started by creating a profile on the Heroku platform. Then, we uploaded our project on the platform and executed the build. By the end, the site was up and running on the server and was publicly accessible. We learned a very important technique for deploying our project. Now, you can share the URL with friends and colleagues to try out your website. You can have a domain name of your choice which you can purchase. Then, you will have to pay for both domain charges and hosting charges.

Try creating another project with different functionalities and deploy it as you deployed this one.

Questions

What changes do you need to make in the settings.py file to make your site deployable?

What is the use of Procfile?

How to get a list of dependencies of your project?

What is Heroku?

What are the different methods to deploy a site on Heroku?

Index

A

account app
forms.py file [162](#)
models.py file [161](#)
settings.py file [165](#)
templates
testing
urls.py file [164](#)
views.py file [163](#)
admin.py file
editing
app, components
admin.py [44](#)
apps.py [44](#)
migrations [45](#)
models.py [44](#)
test.py [44](#)
views.py [45](#)
architecture [20](#)
authentication
versus authorization [158](#)
authorization
versus authentication [158](#)

B

built-in class-based views

about [100](#)

base view [100](#)

basic view

template view [103](#)

built-in field classes

reference link [139](#)

C

class-based views [99](#)

CreatePost view [192](#)

DeletePost view [192](#)

methods [102](#)

need for [100](#)

PostDetail view [191](#)

PostList view [191](#)

UserPosts view [191](#)

configuration code

deploying

Cross-Site Reference Token (CSRF) [136](#)

D

data

adding, to database

editing, in database [50](#)

data entered

fetching, in Django forms

detail view [107](#)

Django
admin page
app, creating
authentication [34](#)
command, executing [39](#)

installing
learning [4](#)
overview [2](#)
project, creating [80](#)
project, enhancing [155](#)
project files, overview
project, settings [34](#)
project, setting up [149](#)
project, testing [79](#)
requirement gathering [67](#)
static files [35](#)
URL [2](#)
Django built-in authentication module
about [159](#)
PermissionMixin model [161](#)
User model [160](#)
Django documentation
reference link [2](#)
Django forms
building
data entered, fetching in
Django's MVT
models [24](#)
templates [27](#)
views [26](#)
Django template

reference link [27](#).

Django Templating Language

about [114](#).

syntax [114](#).

E

elements

adding, to database

deleting, of database [63](#)

manipulating, of database

F

field options

about [86](#)

blank [86](#)

choices [86](#)

db_column [86](#)

default [86](#)

editable [86](#)

help_text [87](#)

null [86](#)

primary_key [87](#)

unique [87](#)

unique_for_date [87](#)

unique_for_month [87](#)

unique_for_year [87](#)

validators [87](#)

verbose_name [87](#)

form field arguments [138](#)

form fields [139](#)

form validation

about

reference link [139](#)

framework

about [3](#)

advantages [3](#)

need for [3](#)

function-based views

about [99](#)

need for [100](#)

functions

in URLconfs [124](#)

G

generic views

about [105](#)

detail view [107](#)

list view [106](#)

genre app

admin.py file [174](#)

models.py file

templates

testing

urls.py file [179](#)

views.py file

get() method [177](#)

get_redirect_url() method [177](#)

H

Heroku
setting up

L

list view [106](#)

M

meta options
about [88](#)
app_label [88](#)
db_table [88](#)
default_manager_name [88](#)
get_latest_by [88](#)
managed [88](#)
ordering [89](#)
order_with_respect_to [89](#)
unique_together [89](#)
verbose_name [90](#)
verbose_name_plural [90](#)
model fields [83](#)
model fields, options
ManyToMany field [94](#)
ManyToOne field [94](#)
OneToOne field [95](#)

model fields, types

AutoField [83](#)

BooleanField [83](#)

CharField [83](#)

DateField [83](#)

DateTimeField [84](#)

DecimalField [84](#)

EmailField [84](#)

FileField [84](#)

FloatField [84](#)

ImageField [84](#)

IntegerField [84](#)

relationship fields [85](#)

SlugField [85](#)

model forms

model instance methods [91](#)

model methods [90](#)

models

about [82](#)

connecting [95](#)

relationship between

models.py file

creating [68](#)

editing

Model-View-Controller (MVC) [20](#)

Model-View-Template (MVT)

o

Object Relational Mapping (ORM)

about [24](#)

overview [54](#)

P

path() method [127](#)

PermissionMixin model [161](#)

post app

forms.py file [195](#)

models.py file

templates

testing

urls.py file [197](#)

views.py file

Q

QuerySet [54](#)

QuerySet, methods

delete [55](#)

exclude [55](#)

filter [55](#)

get [55](#)

latest [55](#)

update [55](#)

QuerySet, operations

bool() [55](#)

iteration [54](#)

len() [54](#)

list() [54](#)

`repr()` [54](#)

slicing [54](#)

R

redirect view

about [103](#)

attributes [104](#)

methods [104](#)

regular expression (regex)

about [125](#)

`path()` method [127](#)

`url/re-path()` method [126](#)

writing, for url-functions [125](#)

`re-path()` method [126](#)

S

settings.py file

arguments and objects

editing

exploring [3Ω](#)

settings.py options

testing

updating

superuser

creating [156](#)

syntax, Django Templating Language

comments [119](#)

filters [118](#)

tags [116](#)

variables [115](#)

T

template inheritance

templates

creating

template view

about [103](#)

redirect view [103](#)

templating language

configuring

U

URLconfs

functions [124](#)

url() function

arguments [123](#)

url() method [126](#)

urls.py file

configuring

User model [160](#)

V

views

creating [69](#)

views, types
about [98](#)
class-based views [99](#)
function-based views [99](#)
virtual environment
about [5](#)
creating
installing [7](#)
using

w

web frameworks
about [4](#)
Angular [4](#)
ASP.NET [4](#)
Flask [4](#)
Play [4](#)
Ruby on Rails [4](#)
Spring [4](#)
Web Server Gateway Interface (WSGI) [32](#)