



1ST EDITION

Django in Production

Expert tips, strategies, and essential frameworks
for writing scalable and maintainable code in Django

ARGHYA SAHA

Django in Production

Expert tips, strategies, and essential frameworks for writing scalable and maintainable code in Django

Arghya Saha



Django in Production

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Rohit Rajkumar

Publishing Product Manager: Jane Dsouza

Book Project Manager: Aishwarya Mohan

Senior Editor: Rashi Dubey

Technical Editor: K Bimala Singha

Copy Editor: Safis Editing

Indexer: Hemangini Bari

Production Designer: Alishon Mendonca

DevRel Marketing Coordinators: Nivedita Pandey and Anamika Singh

First published: April 2024

Production reference: 1070324

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-80461-048-0

www.packtpub.com

To my “Maa,” Anima Saha, and to the memory of my “Baba,” for their constant support in helping me reach where I am today. To my partner, Parul, for being a constant inspiration and motivation.

– Arghya Saha

Contributors

About the author

Arghya (argo) Saha, is a software developer with 8+ years of experience and has been working with Django since 2015. Apart from Django, he is proficient in JavaScript, ReactJS, Node.js, Postgres, AWS, and several other technologies. He has worked with multiple start-ups, such as *Postman* and *HealthifyMe*, among others, to build applications at scale. He currently works at *Abnormal Security* as a senior Site Reliability Engineer to explore his passion in the infrastructure domain.

In his spare time, he writes tech blogs. He is also an adventurous person who has done multiple Himalayan treks and is an endurance athlete with multiple marathons and triathlons under his belt.

This book is dedicated to my Maa and memories of my Baba. I would like to thank my partner, Parul, and all my friends who supported me throughout this journey. I am grateful to Ganesh, Rahul, and everyone who helped me with my book.

About the reviewers

Abdul-Rahman Mustafa Saber Abdul-Aziz is an Egyptian Python backend developer, senior AI instructor, capstone projects lead, and IBM consultant. He graduated from Assiut University in Upper Egypt. He believes that to be great, you must take great responsibility.

I want to thank my family, my friends, and my future wife, who I haven't met yet. I love you so much. I want to thank my father once again; everything I am now is thanks to him. This is the second time that my name has appeared on the Contributors' list of a book. I want to thank Dr. Rania Hafez, Dr. Huda Goda, Dr. Abdul-Rahman Eliwa, and Mr. Ehab, who always pushed me toward progress, and my close friends, Muhammad Adly, Ahmed Fouad, and many others. Thank you all for everything you gave me; you are my family.

Ruben Atinho, a software engineer specializing in backend engineering, explores the vast realms of technology. Proficient in Python, Django, and PostgreSQL, and experienced with Golang, he combines technical expertise with a passion for reading open source code. Beyond the code base, Ruben finds fulfillment in the harmonies of music, insightful articles, captivating books, and the imaginative narratives of anime.

Md Enamul Hasan is a seasoned full stack Python developer with over a decade of professional experience in software and web application development. Based in New York, United States, Enamul is renowned for his expertise in Python, Django, React, and AWS. He has successfully led the development of various software products, including ERP, e-commerce, and data-driven applications.

In addition to his technical proficiency, Enamul has excelled in leadership roles, contributing significantly to project success. He actively shares his knowledge, participating in forums and conferences and showcasing a passion for problem-solving in the ever-evolving software development landscape.

Enamul extends gratitude to the author for the opportunity to review *Django in Production*. His extensive experience in Python Django development adds depth to the review, highlighting the book's value in bridging theory and practice in the dynamic tech industry.

Table of Contents

Preface

xv

Part 1 – Using Django and DRF to Build Modern Web Application

1

Setting Up Django with DRF 3

Technical requirements	4	Creating RESTful API endpoints with DRF	16
Why Django?	4	Best practices for defining RESTful APIs	17
What is available with Django?	5	Best practices to create a REST API with DRF	18
What is the MVT framework?	6	Working with views using DRF	24
Creating a “Hello World” web app with Django and DRF	7	Functional views	25
Creating our Django hello_world project	8	Class-based views	25
Creating our first app in Django	11	Introducing API development tools	27
Linking app views using urls.py	13	Summary	28
Integrating DRF	14		

2

Exploring Django ORM, Models, and Migrations 29

Technical requirements	30	Using models and Django ORM	35
Setting up PostgreSQL with a Django project	31	Adding Django models	35
Creating a PostgreSQL server	31	Basic ORM concepts	36
Configuring Django with PostgreSQL	33	How to get raw queries from ORM	39
		Normalization using Django ORM	40

Exploring on_delete options	44	Keep the default primary key	55
Using model inheritance	44	Use transactions	55
Understanding the crux of Django migrations	46	Avoid generic foreign keys	56
Demystifying migration management commands	46	Use finite state machines (FSMs)	56
Performing database migrations like a pro	48	Break the model into packages	57
Exploring best practices for working with models and ORM	51	Learning about performance optimization	58
Use base models	51	Demystifying performance using explain and analyze	58
Use timezone.now() for any DateTime-related data	52	Using index	59
How to avoid circular dependency in models	52	Using Django ORM like a pro	60
Define __str__ for all models	53	Database connection configuration	64
Use custom model methods	54	Exploring Django Async ORM	66
		Summary	67

3

Serializing Data with DRF 69

Technical requirements	70	Remove default validators from the DRF Serializer class	85
Understanding the basics of DRF Serializers	70	Mastering DRF Serializers	86
Using Model Serializers	71	Using source	86
Creating a new model object	72	Embracing SerializerMethodField	86
Updating existing model Objects	73	Using validators	86
Retrieving data from the Model object instance	74	Using to_internal_value	86
Exploring the Meta class	75	Using to_representation	87
Implementing Serializer relations	78	Use a context argument to pass information	87
Working with nested Serializers	79	Customizing fields	88
Exploring source	80	Passing a custom QuerySet to PrimaryKeyField	88
Exploring SerializerMethodField	81	Building DynamicFieldsSerializer	89
Validating data with serializers	82	Avoiding the N+1 query problem	89
Customizing field-level validation	82	Using Serializers with DRF views	89
Defining a custom field-level validator	83	Working with generic views	90
Performing object-level validation	83	Filtering with SearchFilter and OrderingFilter	91
Defining custom object-level validators	84	Summary	92
The order of the evaluation of validators	84		

4

Exploring Django Admin and Management Commands 93

Technical requirements	94	Renaming admin URLs	100
Exploring Django Admin	94	Using two-factor authentication (2FA) for admin users	100
Creating a superuser in Django	94	Using Custom Admin Paginator	101
Understanding the Django Admin interface	94	Disabling ForeignKey drop-down options	101
Customizing Django Admin	97	Using list_select_related	102
Adding custom fields	97	Overriding get_queryset for performance	102
Using filter_horizontal	97	Adding django-json-widget	102
Using get_queryset	98	Using custom actions	104
Using third-party packages and themes	99	Using permissions for Django Admin	104
Using Django Admin logs	99	Creating custom management commands	105
Optimizing Django Admin for production	100	Summary	107

5

Mastering Django Authentication and Authorization 109

Technical requirements	110	Using DRF token-based authentication	120
Learning the basics of Django authentication	110	Integrating token-based authentication into DRF	120
Customizing the User model	111	Adding DRF token-based authentication to a Django project	121
Using a OneToOneField relationship with the User model	114	Understanding the limitations of token-based authentication of DRF	123
Using Django permissions and groups	114	Learning about third-party token-based authentication packages	124
Using permissions and groups in Django Admin	115	Integrating social login into Django and DRF	125
Creating custom permissions	117	Summary	126
Using Django permissions and groups for an API	118		
Caveats of using permissions	119		

Part 2 – Using the Advanced Concepts of Django

6

Caching, Logging, and Throttling 129

Technical requirements	129	Best practices for throttling in production	139
Caching with Django	130	Logging with Django	139
Using django-cacheops	132	Setting up logging	140
Best practices for caching in production	135	Best practices for logging in production	145
Throttling with Django	136	Summary	145

7

Using Pagination, Django Signals, and Custom Middleware 147

Technical requirements	147	Creating custom signals	152
Paginating responses in Django and DRF	148	Working with signals in production	154
Understanding pagination	148	Working with Django middleware	155
Using pagination in DRF	149	Creating custom middleware	156
Demystifying Django signals	151	Summary	157

8

Using Celery with Django 159

Technical requirements	159	Best practices for using Celery	166
Asynchronous programming in Django	160	Using celery beat with Django	167
Using Celery with Django	160	Summary	169
Integrating Celery and Django	161		
Interfaces of Celery	164		

9

Writing Tests in Django 171

Technical requirements	171	Learning best practices to write tests	189
Introducing the different types of tests in software development	172	Using unit tests more often	189
Unit testing	173	Avoiding time bomb test failures	190
Integration testing	173	Avoiding brittle tests	191
E2E testing	173	Using a reverse function for URL path in tests	191
Setting up tests for Django and DRF	174	Using authentication tests	192
Structuring and naming our test cases	174	Using test tags to group tests	192
Setting up a database for tests	176	Using Postman to create an integration test suite	193
Writing basic tests in DRF	177	Creating different types of tests	193
Writing tests for advanced use cases	179	Avoiding tests	194
Using Django runners	188	Exploring Test-Driven Development	195
		Summary	195

10

Exploring Conventions in Django 197

Technical requirements	198	Configuring Django for production	205
Code structuring for Django projects	198	Setting up CORS	206
Creating files as per functionalities	198	Exploring WSGI	207
Avoiding circular dependencies	200	Summary	208
Creating a “common” app	200		
Working with a settings file for production	201		
Working with exceptions and errors	202		
Using feature flags	204		

Part 3 – Dockerizing and Setting Up a CI Pipeline for Django Application

11

Dockerizing Django Applications 211

Technical requirements	212	Creating a Dockerfile for a Django project	220
Learning the basics of Docker	212	Composing services using docker-compose.yaml	223
Installing Docker	215	Creating a .env file	225
Testing Docker on your local system	216	Accessing environment variables in Django	226
Important commands for Docker	217	Starting a Django application using Docker	227
Working with the requirements.txt file	218	Summary	228

12

Working with Git and CI Pipelines Using Django 229

Technical requirements	229	Working with GitHub Actions for the CI pipeline	240
Using Git efficiently	230	Setting up a CI pipeline for Django using GitHub Actions	240
Branching strategy for Git	231	Recommended GitHub Actions resources	246
Following good practices while using git commit	232	Setting up code review guidelines	247
Tools with Git	233	Context and description	248
Integrating Git hooks into a Django project	234	Short code changes to review	248
Using lefthook	234	Review when the code is ready	248
Using git merge versus git rebase	236	Good code reviewer	249
Performing code release	238	Summary	250
Performing hot-fixing on code	238		
Working with GitHub and GitHub Actions	239		

Part 4 – Deploying and Monitoring Django Applications in Production

13

Deploying Django in AWS 253

Technical requirements	254	Integrating AWS Elastic Beanstalk to deploy Django	264
Learning the basics of AWS	254	Integrating Beanstalk with a basic Django app	264
Creating an account in AWS	255	Deploying a Django application using GitHub Actions in Elastic Beanstalk	279
Identity and Access Management	259	Following the best practices for the AWS infrastructure	282
EC2	261	Best practices for RDS	282
Elastic Load Balancer (ELB)	261	Best practices for ElastiCache	283
Elastic Beanstalk	261	Best practices for Elastic Beanstalk	283
RDS for Postgres	262	Best practices for IAM and security	284
ElastiCache for Redis	262	Summary	284
Security groups and network components	263		
AWS Secrets Manager	263		
Route 53	263		
The AWS Billing console	263		
CloudWatch	263		

14

Monitoring Django Application 285

Technical requirements	285	Integrating APM tools	293
Integrating error monitoring tools	286	Integrating New Relic into the Django project	294
Integrating Rollbar into a Django project	286	Exploring the New Relic dashboard	296
Integrating Rollbar with Slack	288	Creating New Relic alert conditions	299
Best practices while working with error monitoring tools	289	Monitoring AWS EC2 instances with New Relic	301
Integrating uptime monitoring	290	Sending logs from Django to New Relic	301
Adding a health check endpoint	290	Working with metrics and events using NRQL	304
Using BetterStack for uptime monitoring	291	Integrating messaging tools using Slack	305

Handling production incidents better	306	Summary	309
Blameless RCA for incidents	307		
Index			311

Other Books You May Enjoy	324
----------------------------------	------------

Preface

Hey there! As the name suggests, Django in Production is a book to help developers put their application code into production. In today's world, coding has become a profession that people get into after joining a 3–6 month boot camp. With the start-up boom, most of these developers are able to land a job after their boot camp course, since they are able to write code and hack any product together. However, a couple of months into the job, they want to learn about the best practices and understand all the aspects that senior developers in the industry know and perform, but most start-ups don't have many senior developers due to budget and talent scarcity. This book is going to give them a too long; didn't read (TLDR) version of software development best practices, which they need to know to get to the next level.

Who this book is for

This book is for any software developer who understands the basic concepts of Django but now needs some help putting their code to production using the right tools, or someone who does not have enough guidance to know how to do the work the right way. We are assuming you have a basic understanding of how to write code in Django and now want to improve your skills.

What this book covers

Chapter 1, Setting Up Django with DRF, covers the basic project setup of Django and **Django Rest Framework (DRF)**. It will also help you to understand the fundamentals of APIs and how to design a REST API.

Chapter 2, Exploring Django ORM, Models, and Migrations, covers how to integrate Django with a database. Django ORM and migrations are powerful features; we learn about the core concepts and how to use them efficiently in this chapter.

Chapter 3, Serializing Data with DRF, explores the concept of serialization and how developers can use DRF serializers to write better application code.

Chapter 4, Exploring Django Admin and Management Commands, covers the core concepts of Django admin. This chapter covers all the best practices on how to use Django admin and create custom Django management commands.

Chapter 5, Mastering Django Authentication and Authorization, covers the key concepts of authentication and authorization. Django provides authentication and authorization out of the box, and we will explain in detail how developers can use the built-in features of Django and DRF for authentication.

Chapter 6, Caching, Logging, and Throttling, covers all the concepts of caching and how to integrate Redis with Django for caching. Logging is a crucial component of any web application in production and, in this chapter, we will learn how to integrate logging into a Django application.

Chapter 7, Using Pagination, Django Signals, and Custom Middleware, covers all the advanced concepts of Django. Developers can use Django signals to write decoupled code. Django also gives the flexibility to create custom middleware that can help developers to improve features.

Chapter 8, Using Celery with Django, shows how to process asynchronous tasks for web applications. In this chapter, developers will get an understanding of how to integrate Celery into a Django project.

Chapter 9, Writing Tests in Django, covers the core concepts of writing test cases for Django. In this chapter, you will learn the best practices to follow while writing test cases and understand the importance of writing test cases for a project.

Chapter 10, Exploring Conventions in Django, shows all the best practices and conventions that are used while working with Django. This chapter covers a lot of concepts that are opinionated, and you are expected to read this chapter as an outline and pick/learn about concepts by using your own judgment.

Chapter 11, Dockerizing Django Applications, covers how to integrate Docker with a Django application.

Chapter 12, Working with Git and CI Pipelines Using Django, covers the concepts of version control and how to efficiently use Git in a Django project. In this chapter, you will learn how to integrate GitHub Actions to create a CI pipeline.

Chapter 13, Deploying Django in AWS, covers how to deploy Django applications in production using different AWS services. In this chapter, you will learn how to deploy and scale the Django application in production.

Chapter 14, Monitoring Django Applications, covers how to monitor Django applications in production. There are different types of monitoring needed in production, such as error monitoring, application performance monitoring, uptime monitoring, and so on. In this chapter, you will learn how to integrate different tools available on the market to monitor Django applications.

To get the most out of this book

You will need to have a basic understanding of Django and should be comfortable in writing basic Django application code. In this book, we will learn about many of the core concepts of Django and you need to be able to follow those code examples. We will introduce a lot of third-party tools/platforms that may be paid/free, and you are expected to create an account on these platforms and integrate them into the Django project.

Software/hardware covered in the book	Operating system requirements
Python 3.10 and above	Windows, macOS, or Linux
Django 4.x, Django 5.0 and above	
Python packages such as celery, django-fsm, factory-boy, freezetime, django-json-widget, rest_framework	
Docker	
Amazon Web Services (AWS), ElephantSQL, Neon (https://neon.tech), Redis	
Tools such as Rollbar, NewRelic, Better Uptime.	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Django-in-Production>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Since we have specified the `DemoVersion` class, this view would only allow the `v1`, `v2`, and `v3` versions in the URL path; any other version in the path would get a `404` response.”

A block of code is set as follows:

```
urlpatterns = [
    ...
    path('apiview-class/', views.DemoAPIView.as_view())
]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
urlpatterns = [  
    path('hello-world/', views.hello_world),  
    path('demo-version/', views.demo_version),  
    path('custom-version/', views.DemoView.as_view()),  
    path('another-custom-version/', views.AnotherView.as_view())  
]
```

Any command-line input or output is written as follows:

```
celery --app=config beat --loglevel=INFO
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “Click on the **Create New Instance** button.”

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customer@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

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 copyright@packt.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 authors.packtpub.com.

Share Your Thoughts

Once you've read *Django in Production*, we'd love to hear your thoughts! Please visit <https://packt.link/r/1804610488> for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781804610480>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1 – Using Django and DRF to Build Modern Web Application

In the first part of the book, we will get an overview of how to use Django and **Django Rest Framework (DRF)** to create a modern web application. We can expect to learn all the concepts related to Django ORM and DRF serializers, which are crucial to building any modern web application. Django Admin and Authentication are one of the most widely appreciated features of Django. We will learn all the best practices that a developer should know about before using Django and DRF in production.

This part has the following chapters:

- *Chapter 1, Setting Up Django with DRF*
- *Chapter 2, Exploring Django ORM, Models, and Migrations*
- *Chapter 3, Serializing Data with DRF*
- *Chapter 4, Exploring Django Admin and Management Commands*
- *Chapter 5, Mastering Django Authentication and Authorization*

Setting Up Django with DRF

In 2003, the **Django** project was started by developers *Adrian Holovaty* and *Simon Willison* from *World Online*, a newspaper web operation company, and was open sourced and first released in the summer of 2005. When Django was first built, the world was still using dial-up modem internet connections, mobile devices were still not popular, smartphones didn't see the daylight, and people would access web pages through their desktops and laptops. Django was the perfect framework that had all the features needed to build a web application for that age.

Over the last two decades, technology has evolved drastically:

- We have moved from dial-up internet connections to 4G/5G internet connections
- 55% of the world's internet traffic came from mobile devices in 2022 (<https://radar.cloudflare.com/>)

In this book, we shall see how to build a modern web application using Django and deep dive into the core concepts that a developer should know to create a scalable web application for startups. A developer building a product for a startup is expected to be more than just a regular developer writing code in Django; they are expected to develop their code, write tests for the business logic, deploy their applications to the web, and finally keep monitoring the service they have deployed. Here, we will learn how easy it is to develop web applications with Django and the best practices that developers in the industry follow, especially in startups, to make their development cycle easier and faster.

In this first chapter, we shall learn the basics of Django and how to set up a Django project and structure the project folders. Since we shall mostly work with RESTful APIs throughout this book, we will learn about the conventions of the REST API and the crux of setting up a Django project with **Django Rest Framework (DRF)** for creating RESTful APIs. We shall also focus on versioning APIs and how we can implement versioning using DRF. DRF gives us the flexibility to create both functional and class-based views; we shall learn about them in this chapter, along with their pros and cons.

We will cover the following topics:

- Why Django?
- Creating a “Hello World” web app using Django and DRF
- Creating RESTful endpoints with DRF
- Working with views using DRF
- Introducing API development tools

Technical requirements

In this chapter, we shall do the basic project setup and also write our first Hello World app. Though this book is for developers who already know how to write a basic web application, anyone with decent programming skills can pick up this book and learn how to create a scalable Django web application. The following are the skill sets that you should possess to follow this chapter:

- Good Python programming knowledge and familiarity with packages and writing loops, conditional statements, functions, and classes in Python.
- A basic understanding of how web applications work and have written some form of API or web app before.
- Even though we shall try to cover most of the concepts from scratch, having basic knowledge of Django would be great. The Django Girls tutorial is a good resource to learn the basics: <https://tutorial.djangogirls.org/en/>.

You can find the code for this chapter in this book’s GitHub repository: <https://github.com/PacktPublishing/Django-in-Production/tree/main/Chapter01>.

Important note

If you have any doubts about any of the topics mentioned in this or other chapters, feel free to create GitHub issues that specify all the relevant information (<https://github.com/PacktPublishing/Django-in-Production/issues>) or join our *Django in Production* Discord channel and ask us. Here is the invite link for the Discord server, where you can reach me directly: <https://discord.gg/FCrGUfmDyP>.

Why Django?

Django is a web framework based around Python, one of the most popular and easy-to-learn coding languages out there. Since Python is the go-to language for data science and artificial intelligence/machine learning, developers can easily learn Django without having to learn an additional language for building web applications.

Django's tagline, "*Django – The web framework for perfectionists with deadlines,*" proves its commitment to faster and more efficient development, further emphasized by its batteries-included principle that all the basic and widely used functionalities come out of the box with the framework rather than us having to install additional packages. This gives Django an additional advantage over other frameworks, such as Flask.

What is available with Django?

Django has evolved in the last decade and is currently in version 5.x, which has some promising new features, such as asynchronous support. However, the core modules of Django are still the same, with the same principles. When a new developer wants to learn Django, an organization wants to pick Django for their new project, or a startup with limited resources is looking to pick the perfect framework for their tech stack, they want to know why they should learn about Django. To answer this question, we shall learn more about the features of Django.

Let's look at the salient features of Django that the framework provides out of the box:

- In any organization, speed of execution is very important for the success of a product. Django was designed to help developers take applications from the concept phase to the product phase at blazing speed.
- Django takes care of user authentication, content administration, site maps, RSS feeds, and many more fundamental web tasks that developers look for in any framework.
- Security is a serious concern for any organization and Django helps developers avoid common security pitfalls.
- Websites such as Mozilla, Instagram, Disqus, and Pinterest all are built using Django, which makes Django a battle-tested framework that scales.
- Django's versatile framework can be used for different purposes, from content management systems to social networks to scientific computing platforms.

But the question of Django still being relevant is very subjective. Ultimately, it depends upon the use case. We know Django is a good web development framework, however, because more than 55% of the world's internet traffic comes from mobile devices using Android or IOS apps, you may be wondering whether Django is relevant for building features for mobile users? Django as a standalone framework might not be sufficient for today's modern web development where more and more organizations are moving towards API first development, but when integrated with frameworks like Django Rest Framework (DRF), Tastypie, etc, Django becomes the go-to framework for developers.

For start-ups with limited time and resources, it becomes even more crucial to choose a framework where they don't have to build every feature from the ground up, but rather leverage the framework to do most of the heavy lifting.

Let's quickly look a little more at the framework principle that Django uses: the MVT framework.

What is the MVT framework?

Most of us have heard of **MVC frameworks** (*Model-View-Controller*), which represent a paradigm of modern web frameworks where we have the following:

- **Model** represents the data and business logic layer
- **View** represents how the data is presented to the user in the UI/design layout
- **Controller** updates the model and/or view based on the user's input

Django considers the standard names debatable, hence why it has its own interpretation of MVC. Here, we have the following:

- **View** represents *which* data is shown to the end user and not how the data is represented
- **Template** represents *how* the data is represented to the end user
- **Model** represents the data layer

That's why Django follows the **MVT framework** (*Model-View-Template*). But now, the question is, what is the controller in Django? The framework itself is the controller since it handles the whole routing logic using its built-in features.

Important note

You don't need to deep dive into MVT concepts since this concept becomes muscle memory as you write more code in Django.

MVT is a concept where we use templates, but in today's world, most of the products are built for multiple domains such as mobile, IoT, and SaaS platforms. To build products for all these domains, the developer ecosystem has also evolved; now, organizations are moving toward an **API-first** development approach (<https://blog.postman.com/what-is-an-api-first-company/>). This means that APIs are "first-class citizens"; every feature in the product is built with an API-first model, which helps in creating a better client (mobile apps, frontend applications, and so on) and server integration. It involves establishing a contract between the client and the server so that each team can work in parallel without much dependency. Once both teams finish their work, the integration and development cycle of a product becomes much faster with a better developer experience.

The growing use case of mobile device means it is important to build platform-agnostic backend APIs that can be consumed by any client, Android app, iOS app, browser frameworks, and so on. Is Django, as an individual MVT framework, able to serve all these needs? No. The amount of additional effort required to use the out-of-the-box features of Django for creating APIs is similar to reinventing the wheel. That's why most organizations use Django's REST framework, along with Django, to create APIs.

Important note

In this book, instead of focusing on templates and standalone web applications built with Django, we shall focus on creating APIs using Django with DRF. For information on getting started with just Django, one of my favorite resources is the Django Girls tutorial: <https://tutorial.djangogirls.org/en/>.

Now that we have seen what the MVT framework is and how Django is an MVT framework, let's create a basic Hello World web application using Django and set up our project structure and development environment.

Creating a “Hello World” web app with Django and DRF

As mentioned previously, Django is a Python-based web framework, so we need to write the code using the Python programming language. If you are already using Linux or macOS-based systems, then Python comes preinstalled. However, for Windows systems, you have to install it by following the instructions on the official Python website: <https://www.python.org/downloads/>.

We shall also use **virtualenv** as our preferred tool to manage different environments for multiple projects, allowing us to create isolated Python environments.

Important note

We are not going to deep dive into **virtualenv** since we expect you to know how and why we use **virtualenv** for different projects. You can find details about **virtualenv** on its official website: <https://virtualenv.pypa.io/en/latest/index.html>.

First, let's create a virtual environment with the latest Python version (preferably >3.12.0). The following commands will work for Linux/Unix/macOS; for Windows, please check the next section:

```
pip install virtualenv
virtualenv -p python3 v_env
source /path to v_env/v_env/bin/activate
```

Now, we will break down what the code means:

- `pip install virtualenv` installs **virtualenv** on the system. **pip** is the built-in package manager that comes with Python and is already preinstalled on Mac and most Linux environments.
- `virtualenv -p python3 v_env` creates a new virtual environment with the name `v_env` (this is just the name we have given to our virtual environment; you can give another relevant name). The `-p python3` flag is used to tell us which interpreter should be used to create the virtual environment.
- `source /path to v_env/v_env/bin/activate` executes the `activate` script, which loads the virtual Python interpreter as our default Python interpreter in the shell.

Now that the Python virtual environment has been set up, we shall focus on managing the package dependency. To install the latest release of Django, run the following command:

```
pip install Django==5.0.2
```

For Windows systems, download Python 3.12 or higher from <https://www.python.org/downloads/windows/> and install it by following the wizard. Remember to click the **Add python.exe to PATH** checkbox in the installation step.

To verify your Python installation, use the following command in the terminal:

```
C:\Users\argo> python --version
Python 3.12.0
```

Once Python has been installed successfully, you can use the following command to set up a virtual environment and install Django:

```
py -m pip install --user virtualenv
py -m venv venv
.\<path to venv created>\venv\Scripts\activate
// to install Django
pip install Django==5.0.2
```

The explanation for the Windows-specific commands is the same as what we explained for Linux/macOS systems.

Important note

We are not using poetry, PDM, pipenv, or any other dependency and package management tools to avoid overcomplicating the initial setup.

Furthermore, we prefer to use a Docker environment to create more isolation and provide a better developer experience. We shall learn more about Docker in *Chapter 10*.

With the previous command, our local Python and Django development environments are ready. Now, it's time to create our basic Django project.

Creating our Django hello_world project

We all love the `django-admin` command and all the boilerplate code it gives us when we create a new project or application. However, when working on a larger project, the default project structure is not so helpful. This is because when we work with Django in production, we have many other moving parts that need to be incorporated into the project. Project structure and other utilities that are used with a project are always opinionated; what might work for you in your current project might not work in the next project you create a year down the line.

Important note

There are plenty of resources available on the internet that will suggest different project structures. One of my favorites is `django-cookiecutter`. It gives you a lot of tools integrated into the project and gives you a structure that you can follow, but it can be daunting for any new beginner to start since it integrates a lot of third-party tools that you might not use, along with a few configurations that you might not understand. But instead of worrying about that, you can just follow along with this book!

We shall create our own minimalistic project structure and have other tools integrated with our project in incremental steps. First, let’s create our `hello_world` project with Django:

```
mkdir hello_world && cd hello_world
mkdir backend && cd backend
django-admin startproject config .
```

Here, we have created our project folder, `hello_world`, and then created a subfolder called `backend` inside of it. We are using the `backend` folder to keep all the Django-related code; we shall create more folders at the same level as the `backend` subfolder as we learn more about the CI/CD features and incorporate more tools into the project. Finally, we used the Django management command to create our project.

Important note

Note the `.` (dot), which we have appended to the `startproject` command; this tells the Django management command to create the project in the current folder rather than create a separate folder `config` with the project. By default, if you don’t add `.`, then Django will create an additional folder called `config` in which the following project structure will be created. For better understanding, you can test the command with and without `.` to get a clear idea of how it impacts the structure.

After executing these commands, we should be able to see the project structure shown here:



```
tree hello_world/
hello_world/
├── backend
│   ├── config
│   │   ├── __init__.py
│   │   ├── asgi.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   └── manage.py
3 directories, 6 files
```

Figure 1.1: Expected project structure after executing the commands

Now that our project structure is ready, let's run `python manage.py runserver` to verify our Django project. We should see the following output in our shell:

```
python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you
apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
November 04, 2023 - 18:00:26
Django version 4.2, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figure 1.2: The python manage.py runserver command's output in the shell

Please ignore the unapplied migrations warning stating **You have 18 unapplied migrations(s)** displayed in red in the console; we shall discuss this in detail in the next chapter when we learn more about the database, models, and migrations.

Now, go to your browser and open `http://localhost:8000` or `http://127.0.0.1:8000` (if the former fails to load). We shall see the following screen as shown in *Figure 1.3*, which verifies that our server is running successfully:

Please note

You can use `http://localhost:8000` or `http://127.0.0.1:8000` to open the Django project in your browser. If you face any error for `http://localhost:8000`, then please try using `http://127.0.0.1:8000` for all the URLs mentioned in this book.

django

View [release notes](#) for Django 4.2



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



Django Documentation
Topics, references, & how-to's



Tutorial: A Polling App
Get started with Django



Django Community
Connect, get help, or contribute

Figure 1.3: Our Django server running successfully with port 8000

Now, let’s create our first `hello_world` view. To do this, follow these steps:

1. Open the `config/urls.py` file.
2. Add a new view function in `hello_world`.
3. Link the `hello_world` view function to the `hello-world` path.

Our `config/urls.py` file should look like the following code snippet:

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

def hello_world(request):
    return HttpResponse('hello world')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello-world/', hello_world)
]
```

4. Open `http://127.0.0.1:8000/hello-world/` to get the result shown in *Figure 1.4*:



Figure 1.4: `http://127.0.0.1:8000/hello-world/` browser response

So far, we have seen how to create the project folder structure and create our first view in Django. The example we have used is one of the smallest Django project examples that doesn’t involve an app. So, let’s see how we can create apps in Django that can help us manage our project better.

Creating our first app in Django

A Django app can be considered a small package performing one individual functionality in a large project. Django provides management commands to create a new app in a project; these are built-in commands that are used to perform repetitive and complex tasks. The Django community loves management commands since they take away a lot of manual effort and encapsulate a lot of complicated tasks, such as migrations and more. We shall learn more about Django management commands in the following chapters, where we will create a custom management command. However, whenever you see a command followed by `manage.py`, that is a Django management command.

So, let's create a new `demo_app` using the Django management command interface:

```
python manage.py startapp demo_app
```

Running this command will create the folder structure shown here:

```
tree hello_world/
hello_world/
├── backend
│   ├── config
│   │   ├── __init__.py
│   │   ├── asgi.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   └── demo_app
│       ├── __init__.py
│       ├── admin.py
│       ├── apps.py
│       ├── migrations
│       │   └── __init__.py
│       ├── models.py
│       ├── tests.py
│       └── views.py
└── manage.py

5 directories, 13 files
```

Figure 1.5: Project structure with `demo_app` added

We can see that a `demo_app` folder has been created that contains the boilerplate code generated by Django for a new app.

Important note

One important step we must do whenever we create a new app is to tell Django about the new app. Unfortunately, this doesn't happen automatically when you create a new app using the Django management command. It is a manual process where you need to add the details of the new app in the `INSTALLED_APPS` list in the `settings.py` file. Django uses this to identify all the dependency apps added to the project and check for any database-related changes or even register for signals and receivers.

Though adding a new app to the `INSTALLED_APPS` list is not required for us currently, since we are not using `models` for Django to automatically identify any database-related changes, it is still good practice to do so. Our `INSTALLED_APPS` list should look like this:

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    'demo_app',
]
```

Remember to put a comma (,) after every entry of a new app; this is one of the most common mistakes developers make and it causes Django to append two app names into one and generate a syntax error before finally correcting it.

Important note

In Django, third-party app integrations are also done via `INSTALLED_APPS`, so we shall see best practices around how to maintain `INSTALLED_APPS` in the following sections.

Now that we have created a new Django app with the boilerplate code, we can link the app view with `urls.py`.

Linking app views using `urls.py`

In this section, we shall link `views.py`, which was created by the Django management command. `views.py` is used to add business logic to the application endpoints. Just like we added the `hello_world` functional view in the previous section, we can add different functional or class-based views in the `views.py` file.

Let's create a simple `hello_world` functional view in our `demo_app/views.py` file:

```
from django.http import HttpResponse

def hello_world(request, *args, **kwargs):
    return HttpResponse('hello world')
```

As our project grows and the number of apps increases, our main `urls.py` file will become more and more cluttered, with hundreds of URL patterns in a single file. So, it is favorable to break down the main `config/urls.py` file into smaller `urls.py` files for each app, which improves the maintainability of the project.

Now, we will create a new file called `backend/demo_app/urls.py` where we shall add all the routes related to `demo_app`. Subsequently, when we add more apps to the project, we shall create individual `urls.py` files for each app.

Important note

The `urls.py` filename can be anything, but we are keeping this as-is to be consistent with the Django convention.

Add the following code inside the `backend/demo_app/urls.py` file:

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

```
urlpatterns = [  
    path('hello-world/', views.hello_world)  
]
```

Here, we are defining the URL pattern for the `hello-world` path, which links to the basic functional view we created earlier.

Opinionated note

We are using absolute import to import our `demo_app` views. This is a convention we shall follow throughout this book and we also recommend it for other projects. The advantage of using absolute import over relative import is that it is straightforward and clear to read. With just a glance, someone can easily tell what resource has been imported. Also, PEP-8 explicitly recommends using absolute imports.

Now, let's connect the `demo_app/urls.py` file to the main project config/`urls.py` file:

```
from django.contrib import admin  
from django.urls import include  
from django.urls import path  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('demo-app/', include('demo_app.urls'))  
]
```

Next, open `http://127.0.0.1:8000/demo-app/hello-world/` in your browser to make sure our `demo-app` view is linked with Django. You should be able to see `hello world` displayed on the screen, just as we saw earlier in *Figure 1.4*.

So far, we have worked with plain vanilla Django, but now, we'll see how we can integrate DRF into our project.

Integrating DRF

In the API-first world of development, where developers create APIs day in, day out for every feature they build, DRF is a powerful and flexible toolkit for building APIs using Django.

Important note

If you are not familiar with the basics of DRF, we will be going through the basics in this book. However, you can find more information here: <https://www.django-rest-framework.org/tutorial/quickstart/>.