EXPERT INSIGHT

Django 4 By Example

Build powerful and reliable Python web applications from scratch

Foreword by:
Bob Belderbos
Co-Founder of Pybites

Fourth Edition

Antonio Melé

packt

Django 4 By Example

Fourth Edition

Build powerful and reliable Python web applications from scratch

Antonio Melé

‹packt›

BIRMINGHAM—MUMBAI

Django 4 By Example

Fourth Edition

Senior Publishing Product Manager: Manish Nainani
Acquisition Editor – Peer Reviews: Suresh Jain
Project Editor: Amisha Vathare
Content Development Editor: Bhavesh Amin
Copy Editor: Safis Editing
Technical Editor: Aditya Sawant
Proofreader: Safis Editing
Indexer: Sejal Dsilva
Presentation Designer: Pranit Padwal

First published: November 2015
Second edition: May 2018
Third edition: March 2020
Fourth edition: August 2022

Production reference: 2230822

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80181-305-1

www.packt.com

To my sister Paloma.

Foreword

Django: The web framework for perfectionists with deadlines.

I like this tagline because it can be easy for developers to fall prey to perfectionism when having to deliver workable code on time.

There are many great web frameworks out there, but sometimes they assume too much of the developer, for example, how to properly structure a project, find the right plugins and elegantly use existing abstractions.

Django takes most of that decision fatigue away and provides you with so much more. But it's also a big framework, so learning it from scratch can be overwhelming.

I learned Django in 2017, head-on, out of necessity, when we decided it would be our core technology for our Python coding platform (CodeChalleng.es). I forced myself to learn the ins and outs by building a major real-world solution that has served thousands of aspiring and experienced Python developers since its inception.

Somewhere in this journey, I picked up an early edition of this book. It turned out to be a treasure trove. Very close to our hearts at Pybites, it teaches you Django by **building** interesting, real-world applications. Not only that, Antonio brings a lot of real-world experience and knowledge to the table, which shows in how he implements those projects.

And Antonio never misses an opportunity to introduce lesser-known features, for example, optimizing database queries with Postgres, useful packages like django-taggit, social auth using various platforms, (model) managers, inclusion template tags, and much more.

In this new edition, he even added additional schemas, images, and notes in several chapters and moved from jQuery to vanilla JavaScript (nice!)

This book not only covers Django thoroughly, using clean code examples that are well explained, it also explains related technologies which are a must for any Django developer: Django REST Framework, django-debug-toolbar, frontend / JS, and, last but not least, Docker.

More importantly, you'll find many nuances that you'll encounter and best practices you'll need to be an effective Django developer in a professional setting.

Finding a multifaceted resource like this is hard, and I want to thank Antonio for all the hard work he consistently puts into keeping it up to date.

As a Python developer that uses Django a lot, Django by Example has become my **GO TO** guide, an unmissable resource I want to have close by my desk. Every time I come back to this book, I learn new things, even after having read it multiple times and having used Django for a solid five years now.

If you embark on this journey, be prepared to get your hands dirty. It's a practical guide, so brew yourself a good coffee and expect to sink your teeth into a lot of Django code! But that's how we best learn, right? :)

- Bob Belderbos
Co-Founder of Pybites

Contributors

About the author

Antonio Melé is the co-founder and chief technology officer of Nucoro, the fintech platform that allows financial institutions to build, automate, and scale digital wealth management products. Antonio is also CTO of Exo Investing, an AI-driven digital investment platform for the UK market.

Antonio has been developing Django projects since 2006 for clients across several industries. In 2009 Antonio founded Zenx IT, a development company specialized in building digital products. He has been working as a CTO and technology consultant for multiple technology-based startups and he has managed development teams building projects for large digital businesses. Antonio holds an MSc. in Computer Science from ICAI - Universidad Pontificia Comillas, where he mentors early-stage startups. His father inspired his passion for computers and programming.

About the reviewer

Asif Saif Uddin is a software craftsman from Bangladesh. He has a decade-long professional experience working with Python and Django. Besides working for different start-ups and clients, Asif also contributes to some frequently used Python and Django packages. For his open-source contributions, he is now a core maintainer of Celery, oAuthLib, PyJWT, and auditwheel. He is also co-maintainer of several Django and Django REST framework extension packages. He is a voting member of the Django **Software Foundation** (**DSF**) and a contributing/managing member of the **Python Software Foundation** (**PSF**). He has been mentoring many young people to learn Python and Django, both professionally and personally.

*A special thanks to **Karen Stingel** and **Ismir Kullolli** for reading and providing feedback on the book to enhance the content further. Your help is much appreciated!*

Table of Contents

Preface

Django is an open-source Python web framework that encourages rapid development and clean, pragmatic design. It takes care of much of the hassle of web development and presents a relatively shallow learning curve for beginner programmers. Django follows Python's "batteries included" philosophy, shipping with a rich and versatile set of modules that solve common web-development problems. The simplicity of Django, together with its powerful features, makes it attractive to both novice and expert programmers. Django has been designed for simplicity, flexibility, reliability, and scalability.

Nowadays, Django is used by countless start-ups and large organizations such as Instagram, Spotify, Pinterest, Udemy, Robinhood, and Coursera. It is not by coincidence that, over the last few years, Django has consistently been chosen by developers worldwide as one of the most loved web frameworks in Stack Overflow's annual developer survey.

This book will guide you through the entire process of developing professional web applications with Django. The book focuses on explaining how the Django web framework works by building multiple projects from the ground up. This book not only covers the most relevant aspects of the framework but also explains how to apply Django to very diverse real-world situations.

This book not only teaches Django but also presents other popular technologies like PostgreSQL, Redis, Celery, RabbitMQ, and Memcached. You will learn how to integrate these technologies into your Django projects throughout the book to create advanced functionalities and build complex web applications.

Django 4 By Example will walk you through the creation of real-world applications, solving common problems, and implementing best practices, using a step-by-step approach that is easy to follow.

After reading this book, you will have a good understanding of how Django works and how to build full-fledged Python web applications.

Who this book is for

This book should serve as a primer for programmers newly initiated to Django. The book is intended for developers with Python knowledge who wish to learn Django in a pragmatic manner. Perhaps you are completely new to Django, or you already know a little but you want to get the most out of it. This book will help you to master the most relevant areas of the framework by building practical projects from scratch. You need to have familiarity with programming concepts in order to read this book. In addition to basic Python knowledge, some previous knowledge of HTML and JavaScript is assumed.

What this book covers

This book encompasses a range of topics of web application development with Django. The book will guide you through building four different fully-featured web applications, built over the course of 17 chapters:

- A blog application (chapters 1 to 3)
- An image bookmarking website (chapters 4 to 7)
- An online shop (chapters 8 to 11)
- An e-learning platform (chapters 12 to 17)

Each chapter covers several Django features:

Chapter 1, *Building a Blog Application*, will introduce you to the framework through a blog application. You will create the basic blog models, views, templates, and URLs to display blog posts. You will learn how to build QuerySets with the Django **object-relational mapper** (**ORM**), and you will configure the Django administration site.

Chapter 2, *Enhancing Your Blog with Advanced Features*, will teach you how to add pagination to your blog, and how to implement Django class-based views. You will learn to send emails with Django, and handle forms and model forms. You will also implement a comment system for blog posts.

Chapter 3, *Extending Your Blog Application*, explores how to integrate third-party applications. This chapter will guide you through the process of creating a tagging system, and you will learn how to build complex QuerySets to recommend similar posts. The chapter will teach you how to create custom template tags and filters. You will also learn how to use the sitemap framework and create an RSS feed for your posts. You will complete your blog application by building a search engine using PostgreSQL's full-text search capabilities.

Chapter 4, *Building a Social Website*, explains how to build a social website. You will learn how to use the Django authentication framework, and you will extend the user model with a custom profile model. The chapter will teach you how to use the messages framework and you will build a custom authentication backend.

Chapter 5, *Implementing Social Authentication*, covers implementing social authentication with Google, Facebook, and Twitter using OAuth 2 with Python Social Auth. You will learn how to use Django Extensions to run the development server through HTTPS and customize the social authentication pipeline to automate the user profile creation.

Chapter 6, *Sharing Content on Your Website*, will teach you how to transform your social application into an image bookmarking website. You will define many-to-many relationships for models, and you will create a JavaScript bookmarklet that integrates into your project. The chapter will show you how to generate image thumbnails. You will also learn how to implement asynchronous HTTP requests using JavaScript and Django and you will implement infinite scroll pagination.

Chapter 7, *Tracking User Actions*, will show you how to build a follower system for users. You will complete your image bookmarking website by creating a user activity stream application. You will learn how to create generic relations between models and optimize QuerySets. You will work with signals and implement denormalization. You will use Django Debug Toolbar to obtain relevant debug information. Finally, you will integrate Redis into your project to count image views and you will create a ranking of the most viewed images with Redis.

Chapter 8, *Building an Online Shop*, explores how to create an online shop. You will build models for a product catalog, and you will create a shopping cart using Django sessions. You will build a context processor for the shopping cart and will learn how to manage customer orders. The chapter will teach you how to send asynchronous notifications using Celery and RabbitMQ. You will also learn to monitor Celery using Flower.

Chapter 9, *Managing Payments and Orders*, explains how to integrate a payment gateway into your shop. You will integrate Stripe Checkout and receive asynchronous payment notifications in your application. You will implement custom views in the administration site and you will also customize the administration site to export orders to CSV files. You will also learn how to generate PDF invoices dynamically.

Chapter 10, *Extending Your Shop*, will teach you how to create a coupon system to apply discounts to the shopping cart. You will update the Stripe Checkout integration to implement coupon discounts and you will apply coupons to orders. You will use Redis to store products that are usually bought together, and use this information to build a product recommendation engine.

Chapter 11, *Adding Internationalization to Your Shop*, will show you how to add internationalization to your project. You will learn how to generate and manage translation files and translate strings in Python code and Django templates. You will use Rosetta to manage translations and implement per-language URLs. You will learn how to translate model fields using `django-parler` and how to use translations with the ORM. Finally, you will create a localized form field using `django-localflavor`.

Chapter 12, *Building an E-Learning Platform*, will guide you through creating an e-learning platform. You will add fixtures to your project, and create initial models for the content management system. You will use model inheritance to create data models for polymorphic content. You will learn how to create custom model fields by building a field to order objects. You will also implement authentication views for the CMS.

Chapter 13, *Creating a Content Management System*, will teach you how to create a CMS using class-based views and mixins. You will use the Django groups and permissions system to restrict access to views and implement formsets to edit the content of courses. You will also create a drag-and-drop functionality to reorder course modules and their content using JavaScript and Django.

Chapter 14, *Rendering and Caching Content*, will show you how to implement the public views for the course catalog. You will create a student registration system and manage student enrollment on courses. You will create the functionality to render different types of content for the course modules. You will learn how to cache content using the Django cache framework and configure the Memcached and Redis cache backends for your project. Finally, you will learn how to monitor Redis using the administration site.

Chapter 15, *Building an API*, explores building a RESTful API for your project using Django REST framework. You will learn how to create serializers for your models and create custom API views. You will handle API authentication and implement permissions for API views. You will learn how to build API viewsets and routers. The chapter will also teach you how to consume your API using the Requests library.

Chapter 16, *Building a Chat Server*, explains how to use Django Channels to create a real-time chat server for students. You will learn how to implement functionalities that rely on asynchronous communication through WebSockets. You will create a WebSocket consumer with Python and implement a WebSocket client with JavaScript. You will use Redis to set up a channel layer and you will learn how to make your WebSocket consumer fully asynchronous.

Chapter 17, *Going Live*, will show you how to create settings for multiple environments and how to set up a production environment using PostgreSQL, Redis, uWSGI, NGINX, and Daphne with Docker Compose. You will learn how to serve your project securely through HTTPS and use the Django system check framework. The chapter will also teach you how to build a custom middleware and create custom management commands.

To get the most out of this book

- The reader must possess a good working knowledge of Python.
- The reader should be comfortable with HTML and JavaScript.
- It is recommended that the reader goes through parts 1 to 3 of the tutorial in the official Django documentation at `https://docs.djangoproject.com/en/4.1/intro/tutorial01/`.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Django-4-by-example`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781801813051_ColorImages.pdf`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Edit the `models.py` file of the `shop` application.”

A block of code is set as follows:

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

admin.site.register(Post)
```

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

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

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

```
python manage.py runserver
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Fill in the form and click the **Save** button.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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 reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit 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@packtpub.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 `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Django 4 By Example, Fourth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` 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.

1

Building a Blog Application

In this book, you will learn how to build professional Django projects. This chapter will teach you how to build a Django application using the main components of the framework. If you haven't installed Django yet, you will discover how to do so in the first part of this chapter.

Before starting our first Django project, let's take a moment to see what you will learn. This chapter will give you a general overview of the framework. The chapter will guide you through the different major components to create a fully functional web application: models, templates, views, and URLs. After reading it, you will have a good understanding of how Django works and how the different framework components interact.

In this chapter, you will learn the difference between Django projects and applications, and you will learn the most important Django settings. You will build a simple blog application that allows users to navigate through all published posts and read single posts. You will also create a simple administration interface to manage and publish posts. In the next two chapters, you will extend the blog application with more advanced functionalities.

This chapter should serve as a guide to build a complete Django application and shall provide an insight into how the framework works. Don't be concerned if you don't understand all the aspects of the framework. The different framework components will be explored in detail throughout this book.

This chapter will cover the following topics:

- Installing Python
- Creating a Python virtual environment
- Installing Django
- Creating and configuring a Django project
- Building a Django application
- Designing data models
- Creating and applying model migrations
- Creating an administration site for your models

- Working with `QuerySets` and model managers
- Building views, templates, and URLs
- Understanding the Django request/response cycle

Installing Python

Django 4.1 supports Python 3.8, 3.9, and 3.10. In the examples in this book, we will use Python 3.10.6.

If you're using Linux or macOS, you probably have Python installed. If you're using Windows, you can download a Python installer from `https://www.python.org/downloads/windows/`.

Open the command-line shell prompt of your machine. If you are using macOS, open the `/Applications/Utilities` directory in the **Finder**, then double-click **Terminal**. If you are using Windows, open the **Start** menu and type `cmd` into the search box. Then click on the **Command Prompt** application to open it.

Verify that Python is installed on your machine by typing the following command in the shell prompt:

```
python
```

If you see something like the following, then Python is installed on your computer:

```
Python 3.10.6 (v3.10.6:9c7b4bd164, Aug  1 2022, 17:13:48) [Clang 13.0.0
(clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

If your installed Python version is lower than 3.10, or if Python is not installed on your computer, download Python 3.10.6 from `https://www.python.org/downloads/` and follow the instructions to install it. On the download site, you can find Python installers for Windows, macOS, and Linux.

Throughout this book, when Python is referenced in the shell prompt, we will be using `python`, though some systems may require using `python3`. If you are using Linux or macOS and your system's Python is Python 2 you will need to use `python3` to use the Python 3 version you installed.

In Windows, `python` is the Python executable of your default Python installation, whereas `py` is the Python launcher. The Python launcher for Windows was introduced in Python 3.3. It detects what Python versions are installed on your machine and it automatically delegates to the latest version. If you use Windows, it's recommended that you replace `python` with the `py` command. You can read more about the Windows Python launcher at `https://docs.python.org/3/using/windows.html#launcher`.

Creating a Python virtual environment

When you write Python applications, you will usually use packages and modules that are not included in the standard Python library. You may have Python applications that require a different version of the same module. However, only a specific version of a module can be installed system-wide. If you upgrade a module version for an application, you might end up breaking other applications that require an older version of that module.

To address this issue, you can use Python virtual environments. With virtual environments, you can install Python modules in an isolated location rather than installing them globally. Each virtual environment has its own Python binary and can have its own independent set of installed Python packages in its site directories.

Since version 3.3, Python comes with the `venv` library, which provides support for creating lightweight virtual environments. By using the Python `venv` module to create isolated Python environments, you can use different package versions for different projects. Another advantage of using `venv` is that you won't need any administration privileges to install Python packages.

If you are using Linux or macOS, create an isolated environment with the following command:

```
python -m venv my_env
```

Remember to use `python3` instead of `python` if your system comes with Python 2 and you installed Python 3.

If you are using Windows, use the following command instead:

```
py -m venv my_env
```

This will use the Python launcher in Windows.

The previous command will create a Python environment in a new directory named `my_env/`. Any Python libraries you install while your virtual environment is active will go into the `my_env/lib/python3.10/site-packages` directory.

If you are using Linux or macOS, run the following command to activate your virtual environment:

```
source my_env/bin/activate
```

If you are using Windows, use the following command instead:

```
.\my_env\Scripts\activate
```

The shell prompt will include the name of the active virtual environment enclosed in parentheses like this:

```
(my_env) zenx@pc:~ zenx$
```

You can deactivate your environment at any time with the `deactivate` command. You can find more information about `venv` at `https://docs.python.org/3/library/venv.html`.

Installing Django

If you have already installed Django 4.1, you can skip this section and jump directly to the *Creating your first project* section.

Django comes as a Python module and thus can be installed in any Python environment. If you haven't installed Django yet, the following is a quick guide to installing it on your machine.

Installing Django with pip

The `pip` package management system is the preferred method of installing Django. Python 3.10 comes with `pip` preinstalled, but you can find `pip` installation instructions at `https://pip.pypa.io/en/stable/installing/`.

Run the following command at the shell prompt to install Django with `pip`:

```
pip install Django~=4.1.0
```

This will install Django's latest 4.1 version in the Python `site-packages/` directory of your virtual environment.

Now we will check whether Django has been successfully installed. Run the following command in a shell prompt:

```
python -m django --version
```

If you get the output `4.1.X`, Django has been successfully installed on your machine. If you get the message `No module named Django`, Django is not installed on your machine. If you have issues installing Django, you can review the different installation options described in `https://docs.djangoproject.com/en/4.1/intro/install/`.

Django can be installed in different ways. You can find the different installation options at `https://docs.djangoproject.com/en/4.1/topics/install/`.

All Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all requirements at once with the command `pip install -r requirements.txt`.

New features in Django 4

Django 4 introduces a collection of new features, including some backward-incompatible changes, while deprecating other features and eliminating old functionalities. Being a time-based release, there is no drastic change in Django 4, and it is easy to migrate Django 3 applications to the 4.1 release. While Django 3 included for the first time **Asynchronous Server Gateway Interface** (**ASGI**) support, Django 4.0 adds several features such as functional unique constraints for Django models, built-in support for caching data with Redis, a new default timezone implementation using the standard Python package `zoneinfo`, a new `scrypt` password hasher, template-based rendering for forms, as well as other new minor features. Django 4.0 drops support for Python 3.6 and 3.7. It also drops support for PostgreSQL 9.6, Oracle 12.2, and Oracle 18c. Django 4.1 introduces asynchronous handlers for class-based views, an asynchronous ORM interface, new validation of model constraints and new templates for rendering forms. The 4.1 version drops support for PostgreSQL 10 and MariaDB 10.2.

You can read the complete list of changes in the Django 4.0 release notes at `https://docs.djangoproject.com/en/dev/releases/4.0/` and the Django 4.1 release notes at `https://docs.djangoproject.com/en/4.1/releases/4.1/`.

Django overview

Django is a framework consisting of a set of components that solve common web development problems. Django components are loosely coupled, which means they can be managed independently. This helps separate the responsibilities of the different layers of the framework; the database layer knows nothing about how the data is displayed, the template system knows nothing about web requests, and so on.

Django offers maximum code reusability by following the **DRY** (**don't repeat yourself**) principle. Django also fosters rapid development and allows you to use less code by taking advantage of Python's dynamic capabilities, such as introspection.

You can read more about Django's design philosophies at `https://docs.djangoproject.com/en/4.1/misc/design-philosophies/`.

Main framework components

Django follows the **MTV** (**Model-Template-View**) pattern. It is a slightly similar pattern to the well-known **MVC** (**Model-View-Controller**) pattern, where the Template acts as View and the framework itself acts as the Controller.

The responsibilities in the Django MTV pattern are divided as follows:

- **Model** – Defines the logical data structure and is the data handler between the database and the View.
- **Template** – Is the presentation layer. Django uses a plain-text template system that keeps everything that the browser renders.
- **View** – Communicates with the database via the Model and transfers the data to the Template for viewing.

The framework itself acts as the Controller. It sends a request to the appropriate view, according to the Django **URL** configuration.

When developing any Django project, you will always work with models, views, templates, and URLs. In this chapter, you will learn how they fit together.

The Django architecture

Figure 1.1 shows how Django processes requests and how the request/response cycle is managed with the different main Django components: URLs, views, models, and templates:

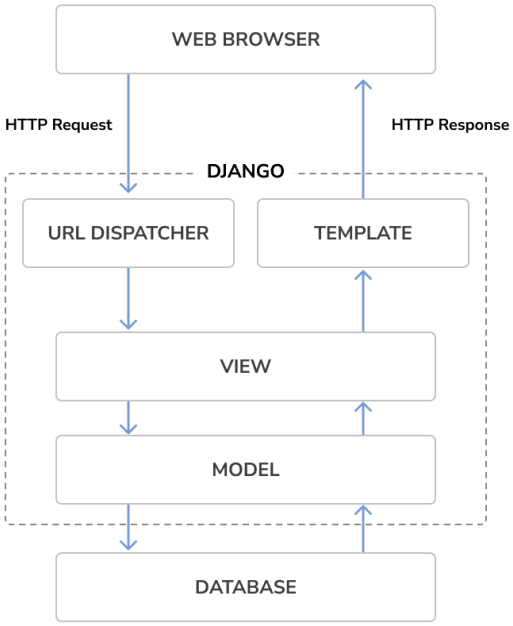


Figure 1.1: The Django architecture

This is how Django handles HTTP requests and generates responses:

1. A web browser requests a page by its URL and the web server passes the HTTP request to Django.
2. Django runs through its configured URL patterns and stops at the first one that matches the requested URL.
3. Django executes the view that corresponds to the matched URL pattern.
4. The view potentially uses data models to retrieve information from the database.
5. Data models provide the data definition and behaviors. They are used to query the database.
6. The view renders a template (usually HTML) to display the data and returns it with an HTTP response.

We will get back to the Django request/response cycle at the end of this chapter in the *The request/response cycle* section.

Django also includes hooks in the request/response process, which are called middleware. Middleware has been intentionally left out of this diagram for the sake of simplicity. You will use middleware in different examples of this book, and you will learn how to create custom middleware in *Chapter 17*, *Going Live*.

Creating your first project

Your first Django project will consist of a blog application. We will start by creating the Django project and a Django application for the blog. We will then create our data models and synchronize them to the database.

Django provides a command that allows you to create an initial project file structure. Run the following command in your shell prompt:

```
django-admin startproject mysite
```

This will create a Django project with the name `mysite`.

Avoid naming projects after built-in Python or Django modules in order to avoid conflicts.

Let's take a look at the generated project structure:

```
mysite/
    manage.py
    mysite/
      __init__.py
      asgi.py
      settings.py
      urls.py
      wsgi.py
```

The outer `mysite/` directory is the container for our project. It contains the following files:

- `manage.py`: This is a command-line utility used to interact with your project. You don't need to edit this file.
- `mysite/`: This is the Python package for your project, which consists of the following files:
  - `__init__.py`: An empty file that tells Python to treat the `mysite` directory as a Python module.
  - `asgi.py`: This is the configuration to run your project as an **Asynchronous Server Gateway Interface** (**ASGI**) application with ASGI-compatible web servers. ASGI is the emerging Python standard for asynchronous web servers and applications.
  - `settings.py`: This indicates settings and configuration for your project and contains initial default settings.
  - `urls.py`: This is the place where your URL patterns live. Each URL defined here is mapped to a view.
  - `wsgi.py`: This is the configuration to run your project as a **Web Server Gateway Interface** (**WSGI**) application with WSGI-compatible web servers.

Applying initial database migrations

Django applications require a database to store data. The `settings.py` file contains the database configuration for your project in the `DATABASES` setting. The default configuration is an SQLite3 database. SQLite comes bundled with Python 3 and can be used in any of your Python applications. SQLite is a lightweight database that you can use with Django for development. If you plan to deploy your application in a production environment, you should use a full-featured database, such as PostgreSQL, MySQL, or Oracle. You can find more information about how to get your database running with Django at `https://docs.djangoproject.com/en/4.1/topics/install/#database-installation`.

Your `settings.py` file also includes a list named `INSTALLED_APPS` that contains common Django applications that are added to your project by default. We will go through these applications later in the *Project settings* section.

Django applications contain data models that are mapped to database tables. You will create your own models in the *Creating the blog data models* section. To complete the project setup, you need to create the tables associated with the models of the default Django applications included in the `INSTALLED_APPS` setting. Django comes with a system that helps you manage database migrations.

Open the shell prompt and run the following commands:

```
cd mysite
python manage.py migrate
```

You will see an output that ends with the following lines:

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sessions.0001_initial... OK
```

The preceding lines are the database migrations that are applied by Django. By applying the initial migrations, the tables for the applications listed in the `INSTALLED_APPS` setting are created in the database.

You will learn more about the `migrate` management command in the *Creating and applying migrations* section of this chapter.

Running the development server

Django comes with a lightweight web server to run your code quickly, without needing to spend time configuring a production server. When you run the Django development server, it keeps checking for changes in your code. It reloads automatically, freeing you from manually reloading it after code changes. However, it might not notice some actions, such as adding new files to your project, so you will have to restart the server manually in these cases.

Start the development server by typing the following command in the shell prompt:

```
python manage.py runserver
```

You should see something like this:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
January 01, 2022 - 10:00:00
Django version 4.0, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Now open `http://127.0.0.1:8000/` in your browser. You should see a page stating that the project is successfully running, as shown in the *Figure 1.2*:

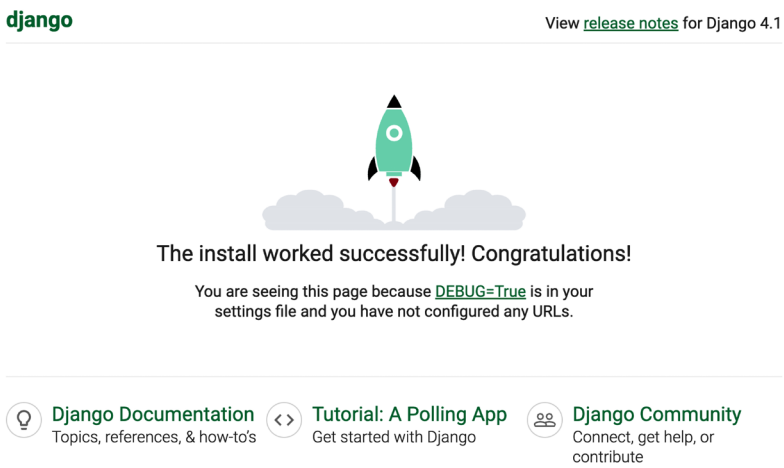


Figure 1.2: The default page of the Django development server

The preceding screenshot indicates that Django is running. If you take a look at your console, you will see the GET request performed by your browser:

```
[01/Jan/2022 17:20:30] "GET / HTTP/1.1" 200 16351
```

Each HTTP request is logged in the console by the development server. Any error that occurs while running the development server will also appear in the console.

You can run the Django development server on a custom host and port or tell Django to load a specific settings file, as follows:

```
python manage.py runserver 127.0.0.1:8001 --settings=mysite.settings
```

When you have to deal with multiple environments that require different configurations, you can create a different settings file for each environment.

This server is only intended for development and is not suitable for production use. To deploy Django in a production environment, you should run it as a WSGI application using a web server, such as Apache, Gunicorn, or uWSGI, or as an ASGI application using a server such as Daphne or Uvicorn. You can find more information on how to deploy Django with different web servers at `https://docs.djangoproject.com/en/4.1/howto/deployment/wsgi/`.

Chapter 17, *Going Live*, explains how to set up a production environment for your Django projects.

Project settings

Let's open the `settings.py` file and take a look at the configuration of the project. There are several settings that Django includes in this file, but these are only part of all the available Django settings. You can see all the settings and their default values at `https://docs.djangoproject.com/en/4.1/ref/settings/`.

Let's review some of the project settings:

- `DEBUG` is a Boolean that turns the debug mode of the project on and off. If it is set to `True`, Django will display detailed error pages when an uncaught exception is thrown by your application. When you move to a production environment, remember that you have to set it to `False`. Never deploy a site into production with `DEBUG` turned on because you will expose sensitive project-related data.
- `ALLOWED_HOSTS` is not applied while debug mode is on or when the tests are run. Once you move your site to production and set `DEBUG` to `False`, you will have to add your domain/host to this setting to allow it to serve your Django site.
- `INSTALLED_APPS` is a setting you will have to edit for all projects. This setting tells Django which applications are active for this site. By default, Django includes the following applications:
    - `django.contrib.admin`: An administration site
    - `django.contrib.auth`: An authentication framework

    - `django.contrib.contenttypes`: A framework for handling content types
    - `django.contrib.sessions`: A session framework
    - `django.contrib.messages`: A messaging framework
    - `django.contrib.staticfiles`: A framework for managing static files
- `MIDDLEWARE` is a list that contains middleware to be executed.
- `ROOT_URLCONF` indicates the Python module where the root URL patterns of your application are defined.
- `DATABASES` is a dictionary that contains the settings for all the databases to be used in the project. There must always be a default database. The default configuration uses an SQLite3 database.
- `LANGUAGE_CODE` defines the default language code for this Django site.
- `USE_TZ` tells Django to activate/deactivate timezone support. Django comes with support for timezone-aware datetimes. This setting is set to `True` when you create a new project using the `startproject` management command.

Don't worry if you don't understand much about what you're seeing here. You will learn more about the different Django settings in the following chapters.

Projects and applications

Throughout this book, you will encounter the terms **project** and **application** over and over. In Django, a project is considered a Django installation with some settings. An application is a group of models, views, templates, and URLs. Applications interact with the framework to provide specific functionalities and may be reused in various projects. You can think of a project as your website, which contains several applications, such as a blog, wiki, or forum, that can also be used by other Django projects.

Figure 1.3 shows the structure of a Django project:

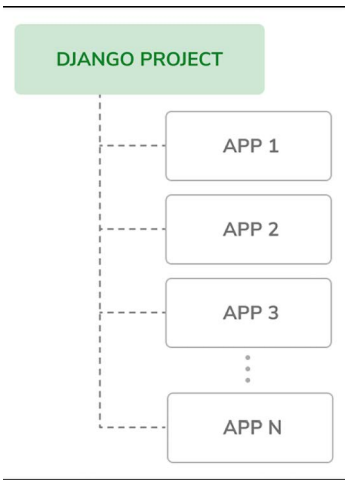


Figure 1.3: The Django project/application structure

Creating an application

Let's create our first Django application. We will build a blog application from scratch.

Run the following command in the shell prompt from the project's root directory:

```
python manage.py startapp blog
```

This will create the basic structure of the application, which will look like this:

```
blog/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

These files are as follows:

- `__init__.py`: An empty file that tells Python to treat the `blog` directory as a Python module.
- `admin.py`: This is where you register models to include them in the Django administration site—using this site is optional.
- `apps.py`: This includes the main configuration of the `blog` application.
- `migrations`: This directory will contain database migrations of the application. Migrations allow Django to track your model changes and synchronize the database accordingly. This directory contains an empty `__init__.py` file.
- `models.py`: This includes the data models of your application; all Django applications need to have a `models.py` file but it can be left empty.
- `tests.py`: This is where you can add tests for your application.
- `views.py`: The logic of your application goes here; each view receives an HTTP request, processes it, and returns a response.

With the application structure ready, we can start building the data models for the blog.

Creating the blog data models

Remember that a Python object is a collection of data and methods. Classes are the blueprint for bundling data and functionality together. Creating a new class creates a new type of object, allowing you to create instances of that type.

A Django model is a source of information and behaviors of your data. It consists of a Python class that subclasses `django.db.models.Model`. Each model maps to a single database table, where each attribute of the class represents a database field. When you create a model, Django will provide you with a practical API to query objects in the database easily.

We will define the database models for our blog application. Then, we will generate the database migrations for the models to create the corresponding database tables. When applying the migrations, Django will create a table for each model defined in the `models.py` file of the application.

Creating the Post model

First, we will define a `Post` model that will allow us to store blog posts in the database.

Add the following lines to the `models.py` file of the `blog` application. The new lines are highlighted in bold:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()

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

This is the data model for blog posts. Posts will have a title, a short label called `slug`, and a body. Let's take a look at the fields of this model:

- `title`: This is the field for the post title. This is a `CharField` field that translates into a `VARCHAR` column in the SQL database.
- `slug`: This is a `SlugField` field that translates into a `VARCHAR` column in the SQL database. A slug is a short label that contains only letters, numbers, underscores, or hyphens. A post with the title *Django Reinhardt: A legend of Jazz* could have a slug like *django-reinhardt-legend-jazz*. We will use the `slug` field to build beautiful, SEO-friendly URLs for blog posts in *Chapter 2, Enhancing Your Blog with Advanced Features*.
- `body`: This is the field for storing the body of the post. This is a `TextField` field that translates into a `TEXT` column in the SQL database.

We have also added a `__str__()` method to the model class. This is the default Python method to return a string with the human-readable representation of the object. Django will use this method to display the name of the object in many places, such as the Django administration site.

If you have been using Python 2.x, note that in Python 3, all strings are natively considered Unicode; therefore, we only use the `__str__()` method. The `__unicode__()` method from Python 2.x is obsolete.

Let's take a look at how the model and its fields will be translated into a database table and columns. The following diagram shows the `Post` model and the corresponding database table that Django will create when we synchronize the model to the database:

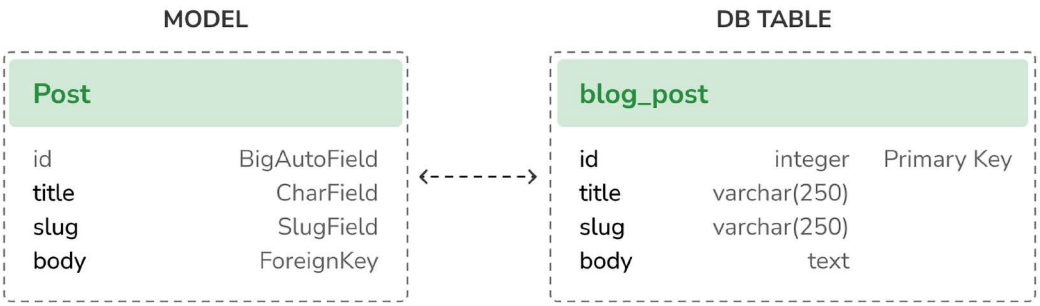


Figure 1.4: Initial Post model and database table correspondence

Django will create a database column for each of the model fields: `title`, `slug`, and `body`. You can see how each field type corresponds to a database data type.

By default, Django adds an auto-incrementing primary key field to each model. The field type for this field is specified in each application configuration or globally in the `DEFAULT_AUTO_FIELD` setting. When creating an application with the `startapp` command, the default value for `DEFAULT_AUTO_FIELD` is `BigAutoField`. This is a 64-bit integer that automatically increments according to available IDs. If you don't specify a primary key for your model, Django adds this field automatically. You can also define one of the model fields to be the primary key by setting `primary_key=True` on it.

We will expand the `Post` model with additional fields and behaviors. Once complete, we will synchronize it to the database by creating a database migration and applying it.

Adding datetime fields

We will continue by adding different datetime fields to the `Post` model. Each post will be published at a specific date and time. Therefore, we need a field to store the publication date and time. We also want to store the date and time when the `Post` object was created, and when it was last modified.

Edit the `models.py` file of the `blog` application to make it look like this. The new lines are highlighted in bold:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
```

```
    updated = models.DateTimeField(auto_now=True)

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

We have added the following fields to the `Post` model:

- `publish`: This is a `DateTimeField` field that translates into a `DATETIME` column in the SQL database. We will use it to store the date and time when the post was published. We use Django's `timezone.now` method as the default value for the field. Note that we imported the `timezone` module to use this method. `timezone.now` returns the current datetime in a timezone-aware format. You can think of it as a timezone-aware version of the standard Python `datetime.now` method.
- `created`: This is a `DateTimeField` field. We will use it to store the date and time when the post was created. By using `auto_now_add`, the date will be saved automatically when creating an object.
- `updated`: This is a `DateTimeField` field. We will use it to store the last date and time when the post was updated. By using `auto_now`, the date will be updated automatically when saving an object.

Defining a default sort order

Blog posts are usually displayed in reverse chronological order (from newest to oldest). We will define a default ordering for our model. The default order will apply when obtaining objects from the database when no order is specified in the query.

Edit the `models.py` file of the `blog` application to make it look like this. The new lines are highlighted in bold:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-publish']

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

We have added a `Meta` class inside the model. This class defines metadata for the model. We use the `ordering` attribute to tell Django that it should sort results by the `publish` field. This ordering will apply by default for database queries when no specific order is provided in the query. We indicate descending order by using a hyphen before the field name, `-publish`. Posts will be returned in reverse chronological order by default.

Adding a database index

Let's define a database index for the `publish` field. This will improve performance for queries filtering or ordering results by this field. We expect many queries to take advantage of this index since we are using the `publish` field to order results by default.

Edit the `models.py` file of the `blog` application and make it look like this. The new lines are highlighted in bold:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

We have added the `indexes` option to the model's `Meta` class. This option allows you to define database indexes for your model, which could comprise one or multiple fields, in ascending or descending order, or functional expressions and database functions. We have added an index for the `publish` field. We use a hyphen before the field name to define the index in descending order. The creation of this index will be included in the database migrations that we will generate later for our blog models.

Index ordering is not supported on MySQL. If you use MySQL for the database, a descending index will be created as a normal index.

You can read more information about how to define indexes for models at `https://docs.djangoproject.com/en/4.1/ref/models/indexes/`.

Activating the application

We need to activate the `blog` application in the project, for Django to keep track of the application and be able to create database tables for its models.

Edit the `settings.py` file and add `blog.apps.BlogConfig` to the `INSTALLED_APPS` setting. It should look like this. The new lines are highlighted in bold:

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

The `BlogConfig` class is the application configuration. Now Django knows that the application is active for this project and will be able to load the application models.

Adding a status field

A common functionality for blogs is to save posts as a draft until ready for publication. We will add a `status` field to our model that will allow us to manage the status of blog posts. We will be using *Draft* and *Published* statuses for posts.

Edit the `models.py` file of the `blog` application to make it look as follows. The new lines are highlighted in bold:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
```

```
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

We have defined the enumeration class `Status` by subclassing `models.TextChoices`. The available choices for the post status are `DRAFT` and `PUBLISHED`. Their respective values are `DF` and `PB`, and their labels or readable names are *Draft* and *Published*.

Django provides enumeration types that you can subclass to define choices simply. These are based on the `enum` object of Python's standard library. You can read more about `enum` at `https://docs.python.org/3/library/enum.html`.

Django enumeration types present some modifications over `enum`. You can learn about those differences at `https://docs.djangoproject.com/en/4.1/ref/models/fields/#enumeration-types`.

We can access `Post.Status.choices` to obtain the available choices, `Post.Status.labels` to obtain the human-readable names, and `Post.Status.values` to obtain the actual values of the choices.

We have also added a new `status` field to the model that is an instance of `CharField`. It includes a `choices` parameter to limit the value of the field to the choices in `Status.choices`. We have also set a default value for the field using the `default` parameter. We use `DRAFT` as the default choice for this field.

It's a good practice to define choices inside the model class and use the enumeration types. This will allow you to easily reference choice labels, values, or names from anywhere in your code. You can import the `Post` model and use `Post.Status.DRAFT` as a reference for the *Draft* status anywhere in your code.

Let's take a look at how to interact with the status choices.

Run the following command in the shell prompt to open the Python shell:

```
python manage.py shell
```

Then, type the following lines:

```
>>> from blog.models import Post
>>> Post.Status.choices
```

You will obtain the enum choices with value-label pairs like this:

```
[('DF', 'Draft'), ('PB', 'Published')]
```

Type the following line:

```
>>> Post.Status.labels
```

You will get the human-readable names of the enum members as follows:

```
['Draft', 'Published']
```

Type the following line:

```
>>> Post.Status.values
```

You will get the values of the enum members as follows. These are the values that can be stored in the database for the `status` field:

```
['DF', 'PB']
```

Type the following line:

```
>>> Post.Status.names
```

You will get the names of the choices like this:

```
['DRAFT', 'PUBLISHED']
```

You can access a specific lookup enumeration member with `Post.Status.PUBLISHED` and you can access its `.name` and `.value` properties as well.

Adding a many-to-one relationship

Posts are always written by an author. We will create a relationship between users and posts that will indicate which user wrote which posts. Django comes with an authentication framework that handles user accounts. The Django authentication framework comes in the `django.contrib.auth` package and contains a `User` model. We will use the `User` model from the Django authentication framework to create a relationship between users and posts.

Edit the `models.py` file of the `blog` application to make it look like this. The new lines are highlighted in bold:

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
```

```
class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)
    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

We have imported the `User` model from the `django.contrib.auth.models` module and we have added an `author` field to the `Post` model. This field defines a many-to-one relationship, meaning that each post is written by a user, and a user can write any number of posts. For this field, Django will create a foreign key in the database using the primary key of the related model.

The `on_delete` parameter specifies the behavior to adopt when the referenced object is deleted. This is not specific to Django; it is an SQL standard. Using `CASCADE`, you specify that when the referenced user is deleted, the database will also delete all related blog posts. You can take a look at all the possible options at `https://docs.djangoproject.com/en/4.1/ref/models/fields/#django.db.models.ForeignKey.on_delete`.

We use `related_name` to specify the name of the reverse relationship, from `User` to `Post`. This will allow us to access related objects easily from a user object by using the `user.blog_posts` notation. We will learn more about this later.

Django comes with different types of fields that you can use to define your models. You can find all field types at `https://docs.djangoproject.com/en/4.1/ref/models/fields/`.

The `Post` model is now complete, and we can now synchronize it to the database. But before that, we need to activate the `blog` application in our Django project.

Creating and applying migrations

Now that we have a data model for blog posts, we need to create the corresponding database table. Django comes with a migration system that tracks the changes made to models and enables them to propagate into the database.

The `migrate` command applies migrations for all applications listed in `INSTALLED_APPS`. It synchronizes the database with the current models and existing migrations.

First, we will need to create an initial migration for our `Post` model.

Run the following command in the shell prompt from the root directory of your project:

```
python manage.py makemigrations blog
```

You should get an output similar to the following one:

```
Migrations for 'blog':
    blog/migrations/0001_initial.py
        - Create model Post
        - Create index blog_post_publish_bb7600_idx on field(s)
          -publish of model post
```

Django just created the `0001_initial.py` file inside the `migrations` directory of the `blog` application. This migration contains the SQL statements to create the database table for the `Post` model and the definition of the database index for the `publish` field.

You can take a look at the file contents to see how the migration is defined. A migration specifies dependencies on other migrations and operations to perform in the database to synchronize it with model changes.

Let's take a look at the SQL code that Django will execute in the database to create the table for your model. The `sqlmigrate` command takes the migration names and returns their SQL without executing it.

Run the following command from the shell prompt to inspect the SQL output of your first migration:

```
python manage.py sqlmigrate blog 0001
```

The output should look as follows:

```
BEGIN;
--
-- Create model Post
--
```

```
CREATE TABLE "blog_post" (
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "title" varchar(250) NOT NULL,
  "slug" varchar(250) NOT NULL,
  "body" text NOT NULL,
  "publish" datetime NOT NULL,
  "created" datetime NOT NULL,
  "updated" datetime NOT NULL,
  "status" varchar(10) NOT NULL,
  "author_id" integer NOT NULL REFERENCES "auth_user" ("id") DEFERRABLE
INITIALLY DEFERRED);
--
-- Create blog_post_publish_bb7600_idx on field(s) -publish of model post
--
CREATE INDEX "blog_post_publish_bb7600_idx" ON "blog_post" ("publish" DESC);
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

The exact output depends on the database you are using. The preceding output is generated for SQLite. As you can see in the output, Django generates the table names by combining the application name and the lowercase name of the model (`blog_post`), but you can also specify a custom database name for your model in the `Meta` class of the model using the `db_table` attribute.

Django creates an auto-incremental `id` column used as the primary key for each model, but you can also override this by specifying `primary_key=True` on one of your model fields. The default `id` column consists of an integer that is incremented automatically. This column corresponds to the `id` field that is automatically added to your model.

The following three database indexes are created:

- An index with descending order on the `publish` column. This is the index we explicitly defined with the `indexes` option of the model's `Meta` class.
- An index on the `slug` column because `SlugField` fields imply an index by default.
- An index on the `author_id` column because `ForeignKey` fields imply an index by default.

Let's compare the `Post` model with its corresponding database `blog_post` table:

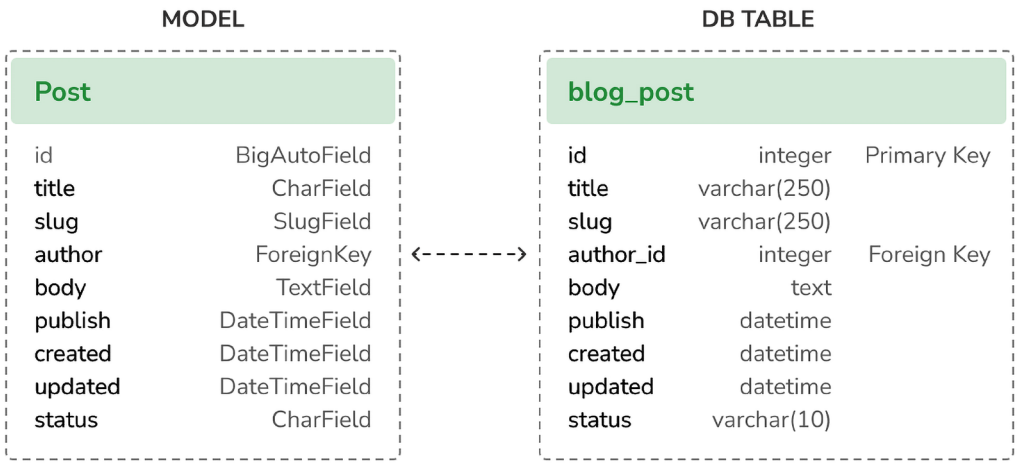


Figure 1.5: Complete Post model and database table correspondence

Figure 1.5 shows how the model fields correspond to database table columns.

Let's sync the database with the new model.

Execute the following command in the shell prompt to apply existing migrations:

```
python manage.py migrate
```

You will get an output that ends with the following line:

```
Applying blog.0001_initial... OK
```

We just applied migrations for the applications listed in `INSTALLED_APPS`, including the `blog` application. After applying the migrations, the database reflects the current status of the models.

If you edit the `models.py` file in order to add, remove, or change the fields of existing models, or if you add new models, you will have to create a new migration using the `makemigrations` command. Each migration allows Django to keep track of model changes. Then, you will have to apply the migration using the `migrate` command to keep the database in sync with your models.

Creating an administration site for models

Now that the `Post` model is in sync with the database, we can create a simple administration site to manage blog posts.

Django comes with a built-in administration interface that is very useful for editing content. The Django site is built dynamically by reading the model metadata and providing a production-ready interface for editing content. You can use it out of the box, configuring how you want your models to be displayed in it.

The `django.contrib.admin` application is already included in the `INSTALLED_APPS` setting, so you don't need to add it.

Creating a superuser

First, you will need to create a user to manage the administration site. Run the following command:

```
python manage.py createsuperuser
```

You will see the following output. Enter your desired username, email, and password, as follows:

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
```

Then you will see the following success message:

```
Superuser created successfully.
```

We just created an administrator user with the highest permissions.

The Django administration site

Start the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/` in your browser. You should see the administration login page, as shown in *Figure 1.6*:

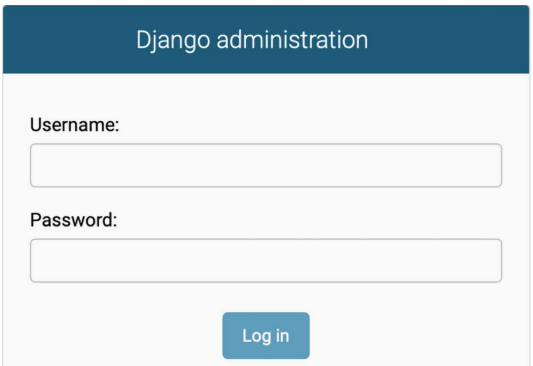


Figure 1.6: The Django administration site login screen

Log in using the credentials of the user you created in the preceding step. You will see the administration site index page, as shown in *Figure 1.7*:

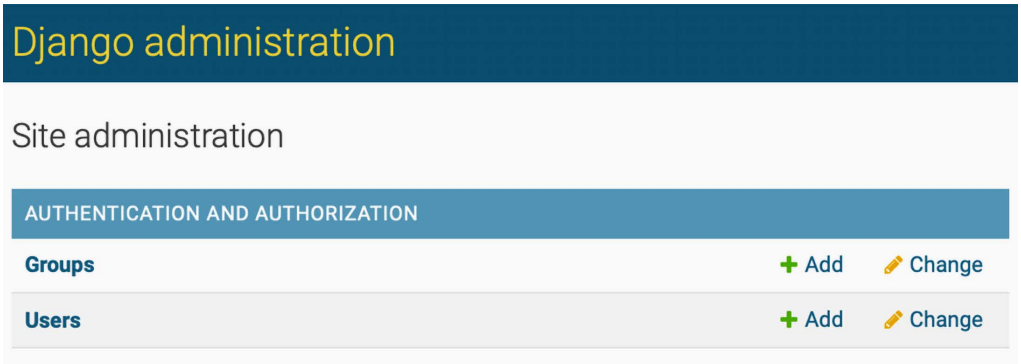


Figure 1.7: The Django administration site index page

The `Group` and `User` models that you can see in the preceding screenshot are part of the Django authentication framework located in `django.contrib.auth`. If you click on **Users**, you will see the user you created previously.

Adding models to the administration site

Let's add your blog models to the administration site. Edit the `admin.py` file of the `blog` application and make it look like this. The new lines are highlighted in bold:

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

admin.site.register(Post)
```

Now reload the administration site in your browser. You should see your `Post` model on the site, as follows:

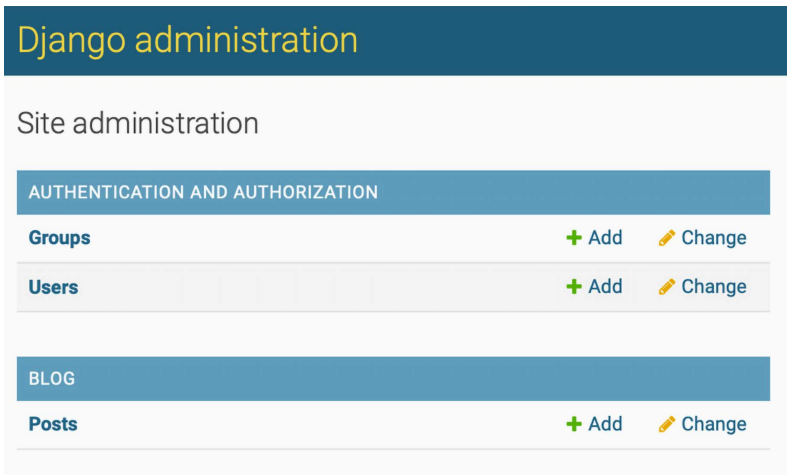


Figure 1.8: The Post model of the blog application included in the Django administration site index page

That was easy, right? When you register a model in the Django administration site, you get a user-friendly interface generated by introspecting your models that allows you to list, edit, create, and delete objects in a simple way.

Click on the **Add** link beside **Posts** to add a new post. You will note the form that Django has generated dynamically for your model, as shown in *Figure 1.9*:

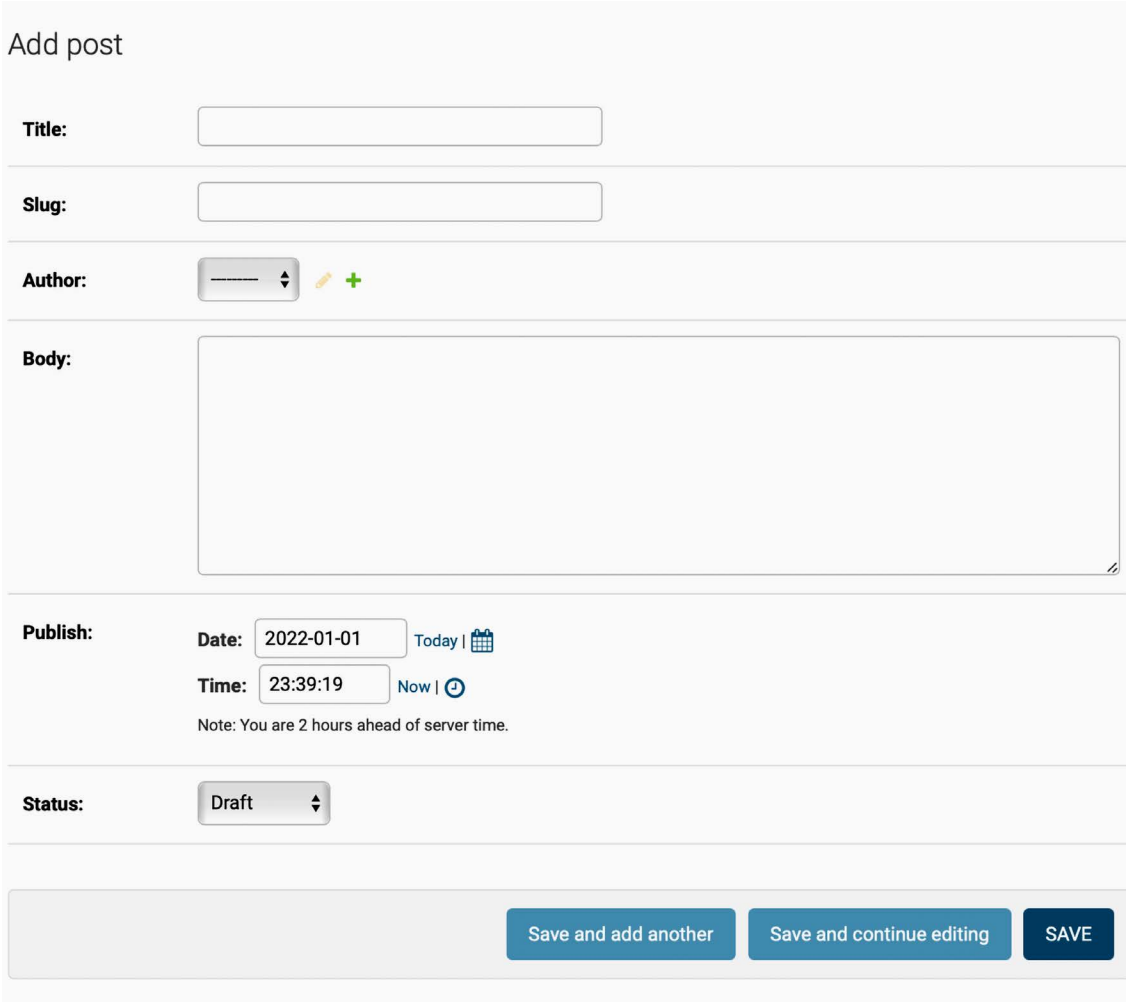


Figure 1.9: The Django administration site edit form for the Post model

Django uses different form widgets for each type of field. Even complex fields, such as the `DateTimeField`, are displayed with an easy interface, such as a JavaScript date picker.

Fill in the form and click on the **SAVE** button. You should be redirected to the post list page with a success message and the post you just created, as shown in *Figure 1.10*:

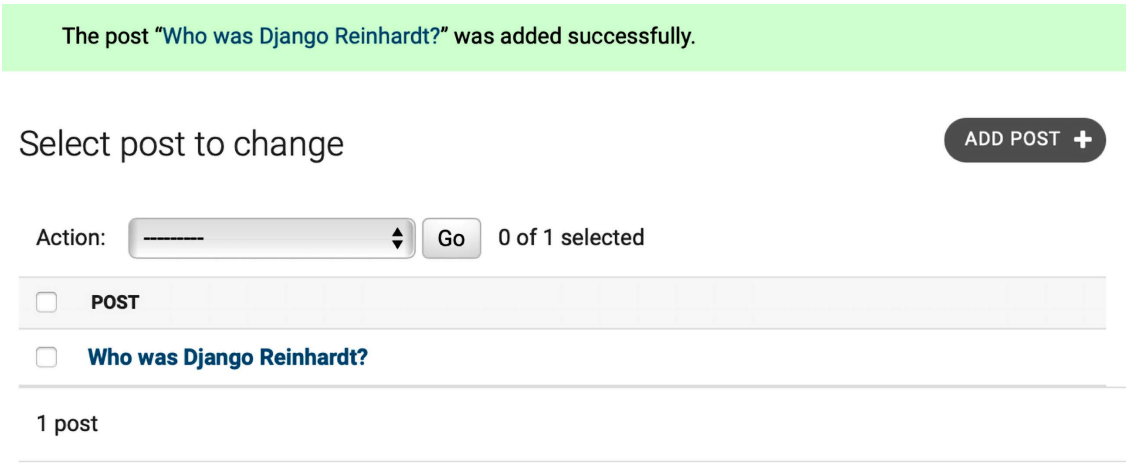


Figure 1.10: The Django administration site list view for the Post model with an added successfully message

Customizing how models are displayed

Now, we will take a look at how to customize the administration site.

Edit the `admin.py` file of your `blog` application and change it, as follows. The new lines are highlighted in bold:

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

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'author', 'publish', 'status']
```

We are telling the Django administration site that the model is registered in the site using a custom class that inherits from `ModelAdmin`. In this class, we can include information about how to display the model on the site and how to interact with it.

The `list_display` attribute allows you to set the fields of your model that you want to display on the administration object list page. The `@admin.register()` decorator performs the same function as the `admin.site.register()` function that you replaced, registering the `ModelAdmin` class that it decorates.

Let's customize the `admin` model with some more options.

Edit the `admin.py` file of your `blog` application and change it, as follows. The new lines are highlighted in bold:

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

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'author', 'publish', 'status']
    list_filter = ['status', 'created', 'publish', 'author']
    search_fields = ['title', 'body']
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ['author']
    date_hierarchy = 'publish'
    ordering = ['status', 'publish']
```

Return to your browser and reload the post list page. Now, it will look like this:

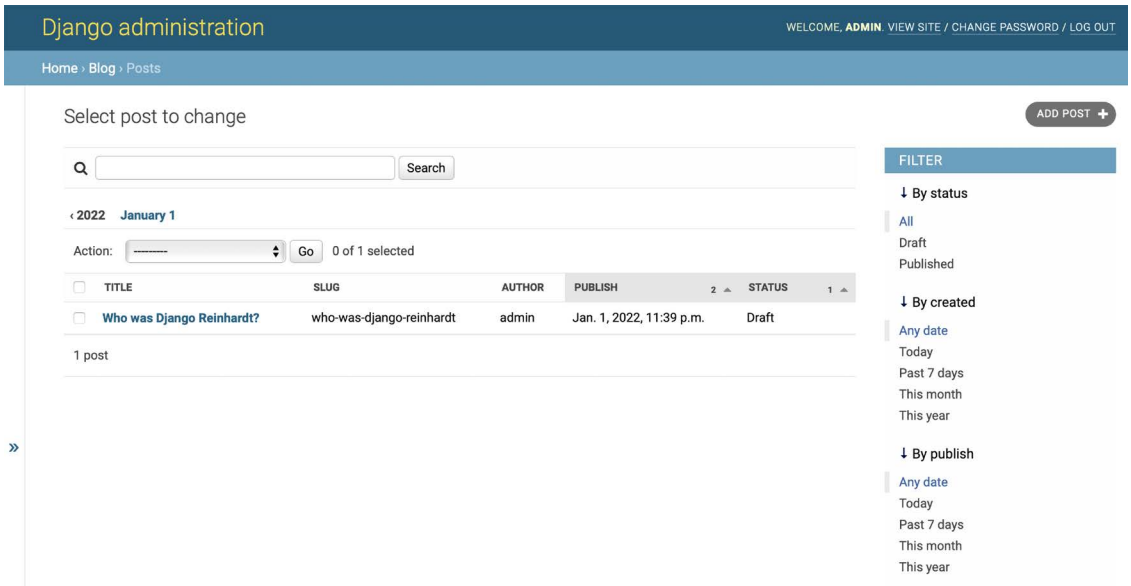


Figure 1.11: The Django administration site custom list view for the Post model

You can see that the fields displayed on the post list page are the ones we specified in the `list_display` attribute. The list page now includes a right sidebar that allows you to filter the results by the fields included in the `list_filter` attribute.

A search bar has appeared on the page. This is because we have defined a list of searchable fields using the `search_fields` attribute. Just below the search bar, there are navigation links to navigate through a date hierarchy; this has been defined by the `date_hierarchy` attribute. You can also see that the posts are ordered by **STATUS** and **PUBLISH** columns by default. We have specified the default sorting criteria using the `ordering` attribute.

Next, click on the **ADD POST** link. You will also note some changes here. As you type the title of a new post, the `slug` field is filled in automatically. You have told Django to prepopulate the `slug` field with the input of the `title` field using the `prepopulated_fields` attribute:

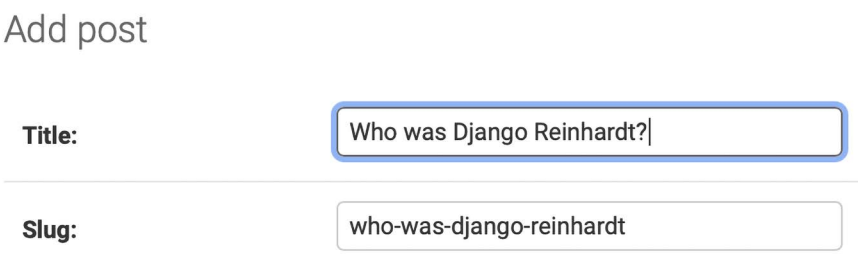


Figure 1.12: The slug model is now automatically prepopulated as you type in the title

Also, the `author` field is now displayed with a lookup widget, which can be much better than a drop-down select input when you have thousands of users. This is achieved with the `raw_id_fields` attribute and it looks like this:

Figure 1.13: The widget to select related objects for the author field of the Post model

With a few lines of code, we have customized the way the model is displayed on the administration site. There are plenty of ways to customize and extend the Django administration site; you will learn more about this later in this book.

You can find more information about the Django administration site at `https://docs.djangoproject.com/en/4.1/ref/contrib/admin/`.

Working with QuerySets and managers

Now that we have a fully functional administration site to manage blog posts, it is a good time to learn how to read and write content to the database programmatically.

The Django **object-relational mapper** (**ORM**) is a powerful database abstraction API that lets you create, retrieve, update, and delete objects easily. An ORM allows you to generate SQL queries using the object-oriented paradigm of Python. You can think of it as a way to interact with your database in pythonic fashion instead of writing raw SQL queries.

The ORM maps your models to database tables and provides you with a simple pythonic interface to interact with your database. The ORM generates SQL queries and maps the results to model objects. The Django ORM is compatible with MySQL, PostgreSQL, SQLite, Oracle, and MariaDB.

Remember that you can define the database of your project in the `DATABASES` setting of your project's `settings.py` file. Django can work with multiple databases at a time, and you can program database routers to create custom data routing schemes.

Once you have created your data models, Django gives you a free API to interact with them. You can find the data model reference of the official documentation at `https://docs.djangoproject.com/en/4.1/ref/models/`.

The Django ORM is based on QuerySets. A QuerySet is a collection of database queries to retrieve objects from your database. You can apply filters to QuerySets to narrow down the query results based on given parameters.

Creating objects

Run the following command in the shell prompt to open the Python shell:

```
python manage.py shell
```

Then, type the following lines:

```
>>> from django.contrib.auth.models import User
>>> from blog.models import Post
>>> user = User.objects.get(username='admin')
>>> post = Post(title='Another post',
...             slug='another-post',
...             body='Post body.',
...             author=user)
>>> post.save()
```

Let's analyze what this code does.

First, we are retrieving the `user` object with the username `admin`:

```
user = User.objects.get(username='admin')
```

The `get()` method allows you to retrieve a single object from the database. Note that this method expects a result that matches the query. If no results are returned by the database, this method will raise a `DoesNotExist` exception, and if the database returns more than one result, it will raise a `MultipleObjectsReturned` exception. Both exceptions are attributes of the model class that the query is being performed on.

Then, we are creating a `Post` instance with a custom title, slug, and body, and set the user that we previously retrieved as the author of the post:

```
post = Post(title='Another post', slug='another-post', body='Post body.',
author=user)
```

This object is in memory and not persisted to the database; we created a Python object that can be used during runtime but that is not saved into the database.

Finally, we are saving the `Post` object to the database using the `save()` method:

```
post.save()
```

The preceding action performs an `INSERT` SQL statement behind the scenes.

We created an object in memory first and then persisted it to the database. You can also create the object and persist it into the database in a single operation using the `create()` method, as follows:

```
Post.objects.create(title='One more post',
                    slug='one-more-post',
                    body='Post body.',
                    author=user)
```

Updating objects

Now, change the title of the post to something different and save the object again:

```
>>> post.title = 'New title'
>>> post.save()
```

This time, the `save()` method performs an `UPDATE` SQL statement.

The changes you make to a model object are not persisted to the database until you call the `save()` method.

Retrieving objects

You already know how to retrieve a single object from the database using the `get()` method. We accessed this method using `Post.objects.get()`. Each Django model has at least one manager, and the default manager is called `objects`. You get a `QuerySet` object using your model manager.

To retrieve all objects from a table, we use the `all()` method on the default objects manager, like this:

```
>>> all_posts = Post.objects.all()
```

This is how we create a `QuerySet` that returns all objects in the database. Note that this `QuerySet` has not been executed yet. Django `QuerySets` are *lazy*, which means they are only evaluated when they are forced to. This behavior makes `QuerySets` very efficient. If you don't assign the `QuerySet` to a variable but instead write it directly on the Python shell, the SQL statement of the `QuerySet` is executed because you are forcing it to generate output:

```
>>> Post.objects.all()
<QuerySet [<Post: Who was Django Reinhardt?>, <Post: New title>]>
```

Using the filter() method

To filter a `QuerySet`, you can use the `filter()` method of the manager. For example, you can retrieve all posts published in the year 2022 using the following `QuerySet`:

```
>>> Post.objects.filter(publish__year=2022)
```

You can also filter by multiple fields. For example, you can retrieve all posts published in 2022 by the author with the username `admin`:

```
>>> Post.objects.filter(publish__year=2022, author__username='admin')
```

This equates to building the same `QuerySet` chaining multiple filters:

```
>>> Post.objects.filter(publish__year=2022) \
>>>              .filter(author__username='admin')
```

Queries with field lookup methods are built using two underscores, for example, `publish__year`, but the same notation is also used for accessing fields of related models, such as `author__username`.

Using exclude()

You can exclude certain results from your `QuerySet` using the `exclude()` method of the manager. For example, you can retrieve all posts published in 2022 whose titles don't start with `Why`:

```
>>> Post.objects.filter(publish__year=2022) \
>>>              .exclude(title__startswith='Why')
```

Using order_by()

You can order results by different fields using the `order_by()` method of the manager. For example, you can retrieve all objects ordered by their `title`, as follows:

```
>>> Post.objects.order_by('title')
```

Ascending order is implied. You can indicate descending order with a negative sign prefix, like this:

```
>>> Post.objects.order_by('-title')
```

Deleting objects

If you want to delete an object, you can do it from the object instance using the `delete()` method:

```
>>> post = Post.objects.get(id=1)
>>> post.delete()
```

Note that deleting objects will also delete any dependent relationships for `ForeignKey` objects defined with `on_delete` set to `CASCADE`.

When QuerySets are evaluated

Creating a QuerySet doesn't involve any database activity until it is evaluated. QuerySets usually return another unevaluated QuerySet. You can concatenate as many filters as you like to a QuerySet, and you will not hit the database until the QuerySet is evaluated. When a QuerySet is evaluated, it translates into an SQL query to the database.

QuerySets are only evaluated in the following cases:

- The first time you iterate over them
- When you slice them, for instance, `Post.objects.all()[:3]`
- When you pickle or cache them
- When you call `repr()` or `len()` on them
- When you explicitly call `list()` on them
- When you test them in a statement, such as `bool()`, `or`, `and`, or `if`

Creating model managers

The default manager for every model is the `objects` manager. This manager retrieves all the objects in the database. However, we can define custom managers for models.

Let's create a custom manager to retrieve all posts that have a `PUBLISHED` status.

There are two ways to add or customize managers for your models: you can add extra manager methods to an existing manager or create a new manager by modifying the initial `QuerySet` that the manager returns. The first method provides you with a `QuerySet` notation like `Post.objects.my_manager()`, and the latter provides you with a `QuerySet` notation like `Post.my_manager.all()`.

We will choose the second method to implement a manager that will allow us to retrieve posts using the notation `Post.published.all()`.

Edit the `models.py` file of your `blog` application to add the custom manager as follows. The new lines are highlighted in bold:

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
                      .filter(status=Post.Status.PUBLISHED)

class Post(models.Model):

    # model fields
    # ...

    objects = models.Manager() # The default manager.
    published = PublishedManager() # Our custom manager.

    class Meta:
        ordering = ['-publish']

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

The first manager declared in a model becomes the default manager. You can use the `Meta` attribute `default_manager_name` to specify a different default manager. If no manager is defined in the model, Django automatically creates the `objects` default manager for it. If you declare any managers for your model, but you want to keep the `objects` manager as well, you have to add it explicitly to your model. In the preceding code, we have added the default `objects` manager and the `published` custom manager to the `Post` model.

The `get_queryset()` method of a manager returns the `QuerySet` that will be executed. We have overridden this method to build a custom `QuerySet` that filters posts by their status and returns a successive `QuerySet` that only includes posts with the `PUBLISHED` status.

We have now defined a custom manager for the `Post` model. Let's test it!

Start the development server again with the following command in the shell prompt:

```
python manage.py shell
```

Now, you can import the `Post` model and retrieve all published posts whose title starts with `Who`, executing the following `QuerySet`:

```
>>> from blog.models import Post
>>> Post.published.filter(title__startswith='Who')
```

To obtain results for this `QuerySet`, make sure to set the `status` field to `PUBLISHED` in the `Post` object whose `title` starts with the string *Who*.

Building list and detail views

Now that you understand how to use the ORM, you are ready to build the views of the `blog` application. A Django view is just a Python function that receives a web request and returns a web response. All the logic to return the desired response goes inside the view.

First, you will create your application views, then you will define a URL pattern for each view, and finally, you will create HTML templates to render the data generated by the views. Each view will render a template, passing variables to it, and will return an HTTP response with the rendered output.

Creating list and detail views

Let's start by creating a view to display the list of posts.

Edit the `views.py` file of the `blog` application and make it look like this. The new lines are highlighted in bold:

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

def post_list(request):
```

```
    posts = Post.published.all()
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts})
```

This is our very first Django view. The `post_list` view takes the `request` object as the only parameter. This parameter is required by all views.

In this view, we retrieve all the posts with the `PUBLISHED` status using the `published` manager that we created previously.

Finally, we use the `render()` shortcut provided by Django to render the list of posts with the given template. This function takes the `request` object, the template path, and the context variables to render the given template. It returns an `HttpResponse` object with the rendered text (normally HTML code).

The `render()` shortcut takes the request context into account, so any variable set by the template context processors is accessible by the given template. Template context processors are just callables that set variables into the context. You will learn how to use context processors in *Chapter 4*, *Building a Social Website*.

Let's create a second view to display a single post. Add the following function to the `views.py` file:

```
from django.http import Http404

def post_detail(request, id):
    try:
        post = Post.published.get(id=id)
    except Post.DoesNotExist:
        raise Http404("No Post found.")

    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

This is the post detail view. This view takes the `id` argument of a post. In the view, we try to retrieve the `Post` object with the given `id` by calling the `get()` method on the default `objects` manager. We raise an `Http404` exception to return an HTTP 404 error if the model `DoesNotExist` exception is raised, because no result is found.

Finally, we use the `render()` shortcut to render the retrieved post using a template.

Using the get_object_or_404 shortcut

Django provides a shortcut to call `get()` on a given model manager and raises an `Http404` exception instead of a `DoesNotExist` exception when no object is found.

Edit the `views.py` file to import the `get_object_or_404` shortcut and change the `post_detail` view as follows. The new code is highlighted in bold:

```
from django.shortcuts import render, get_object_or_404

# ...

def post_detail(request, id):
    post = get_object_or_404(Post,
                             id=id,
                             status=Post.Status.PUBLISHED)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

In the detail view, we now use the `get_object_or_404()` shortcut to retrieve the desired post. This function retrieves the object that matches the given parameters or an HTTP 404 (not found) exception if no object is found.

Adding URL patterns for your views

URL patterns allow you to map URLs to views. A URL pattern is composed of a string pattern, a view, and, optionally, a name that allows you to name the URL project-wide. Django runs through each URL pattern and stops at the first one that matches the requested URL. Then, Django imports the view of the matching URL pattern and executes it, passing an instance of the `HttpRequest` class and the keyword or positional arguments.

Create a `urls.py` file in the directory of the `blog` application and add the following lines to it:

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

app_name = 'blog'

urlpatterns = [
    # post views
    path('', views.post_list, name='post_list'),
    path('<int:id>/', views.post_detail, name='post_detail'),
]
```

In the preceding code, you define an application namespace with the `app_name` variable. This allows you to organize URLs by application and use the name when referring to them. You define two different patterns using the `path()` function. The first URL pattern doesn't take any arguments and is mapped to the `post_list` view. The second pattern is mapped to the `post_detail` view and takes only one argument `id`, which matches an integer, set by the path converter `int`.

You use angle brackets to capture the values from the URL. Any value specified in the URL pattern as `<parameter>` is captured as a string. You use path converters, such as `<int:year>`, to specifically match and return an integer. For example `<slug:post>` would specifically match a slug (a string that can only contain letters, numbers, underscores, or hyphens). You can see all path converters provided by Django at `https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters`.

If using `path()` and converters isn't sufficient for you, you can use `re_path()` instead to define complex URL patterns with Python regular expressions. You can learn more about defining URL patterns with regular expressions at `https://docs.djangoproject.com/en/4.1/ref/urls/#django.urls.re_path`. If you haven't worked with regular expressions before, you might want to take a look at the *Regular Expression HOWTO* located at `https://docs.python.org/3/howto/regex.html` first.

Creating a `urls.py` file for each application is the best way to make your applications reusable by other projects.

Next, you have to include the URL patterns of the `blog` application in the main URL patterns of the project.

Edit the `urls.py` file located in the `mysite` directory of your project and make it look like the following. The new code is highlighted in bold:

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

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

The new URL pattern defined with `include` refers to the URL patterns defined in the `blog` application so that they are included under the `blog/` path. You include these patterns under the namespace `blog`. Namespaces have to be unique across your entire project. Later, you will refer to your blog URLs easily by using the namespace followed by a colon and the URL name, for example, `blog:post_list` and `blog:post_detail`. You can learn more about URL namespaces at `https://docs.djangoproject.com/en/4.1/topics/http/urls/#url-namespaces`.

Creating templates for your views

You have created views and URL patterns for the `blog` application. URL patterns map URLs to views, and views decide which data gets returned to the user. Templates define how the data is displayed; they are usually written in HTML in combination with the Django template language. You can find more information about the Django template language at `https://docs.djangoproject.com/en/4.1/ref/templates/language/`.

Let's add templates to your application to display posts in a user-friendly manner.

Create the following directories and files inside your `blog` application directory:

```
templates/
    blog/
        base.html
        post/
            list.html
            detail.html
```

The preceding structure will be the file structure for your templates. The `base.html` file will include the main HTML structure of the website and divide the content into the main content area and a sidebar. The `list.html` and `detail.html` files will inherit from the `base.html` file to render the blog post list and detail views, respectively.

Django has a powerful template language that allows you to specify how data is displayed. It is based on *template tags*, *template variables*, and *template filters*:

- Template tags control the rendering of the template and look like `{% tag %}`
- Template variables get replaced with values when the template is rendered and look like `{{ variable }}`
- Template filters allow you to modify variables for display and look like `{{ variable|filter }}`

You can see all built-in template tags and filters at `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/`.

Creating a base template

Edit the `base.html` file and add the following code:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
```

```
    <p>This is my blog.</p>
  </div>
</body>
</html>
```

`{% load static %}` tells Django to load the `static` template tags that are provided by the `django.contrib.staticfiles` application, which is contained in the `INSTALLED_APPS` setting. After loading them, you can use the `{% static %}` template tag throughout this template. With this template tag, you can include the static files, such as the `blog.css` file, which you will find in the code of this example under the `static/` directory of the `blog` application. Copy the `static/` directory from the code that comes along with this chapter into the same location as your project to apply the CSS styles to the templates. You can find the directory's contents at `https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter01/mysite/blog/static`.

You can see that there are two `{% block %}` tags. These tell Django that you want to define a block in that area. Templates that inherit from this template can fill in the blocks with content. You have defined a block called `title` and a block called `content`.

Creating the post list template

Let's edit the `post/list.html` file and make it look like the following:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{% url 'blog:post_detail' post.id %}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

With the `{% extends %}` template tag, you tell Django to inherit from the `blog/base.html` template. Then, you fill the `title` and `content` blocks of the base template with content. You iterate through the posts and display their title, date, author, and body, including a link in the title to the detail URL of the post. We build the URL using the `{% url %}` template tag provided by Django.

This template tag allows you to build URLs dynamically by their name. We use `blog:post_detail` to refer to the `post_detail` URL in the `blog` namespace. We pass the required `post.id` parameter to build the URL for each post.

Always use the `{% url %}` template tag to build URLs in your templates instead of writing hardcoded URLs. This will make your URLs more maintainable.

In the body of the post, we apply two template filters: `truncatewords` truncates the value to the number of words specified, and `linebreaks` converts the output into HTML line breaks. You can concatenate as many template filters as you wish; each one will be applied to the output generated by the preceding one.

Accessing our application

Open the shell and execute the following command to start the development server:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/blog/` in your browser; you will see everything running. Note that you need to have some posts with the `PUBLISHED` status to show them here. You should see something like this:

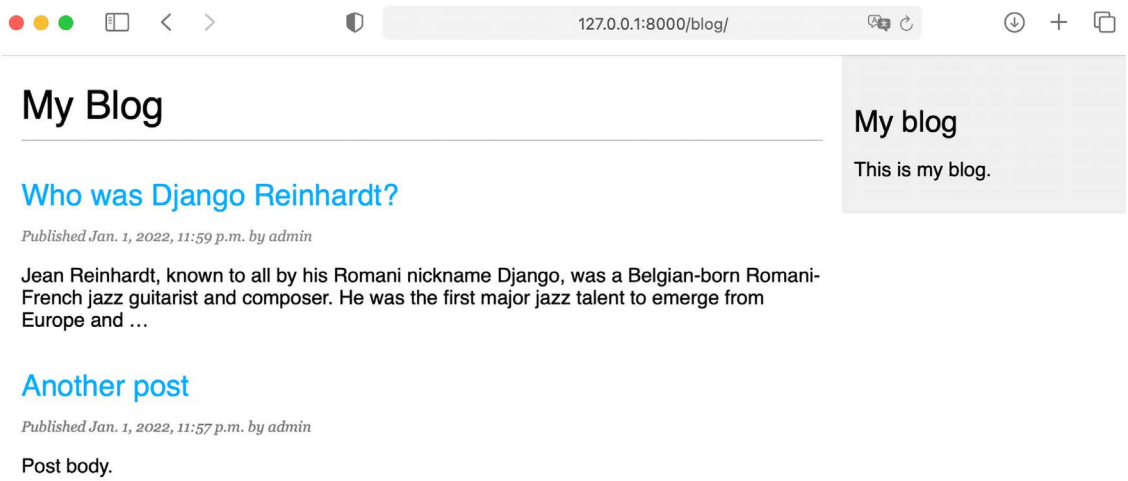


Figure 1.14: The page for the post list view

Creating the post detail template

Next, edit the `post/detail.html` file:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
{% endblock %}
```

Next, you can return to your browser and click on one of the post titles to take a look at the detail view of the post. You should see something like this:

Figure 1.15: The page for the post's detail view

Take a look at the URL—it should include the auto-generated post ID like `/blog/1/`.

The request/response cycle

Let's review the request/response cycle of Django with the application we built. The following schema shows a simplified example of how Django processes HTTP requests and generates HTTP responses:

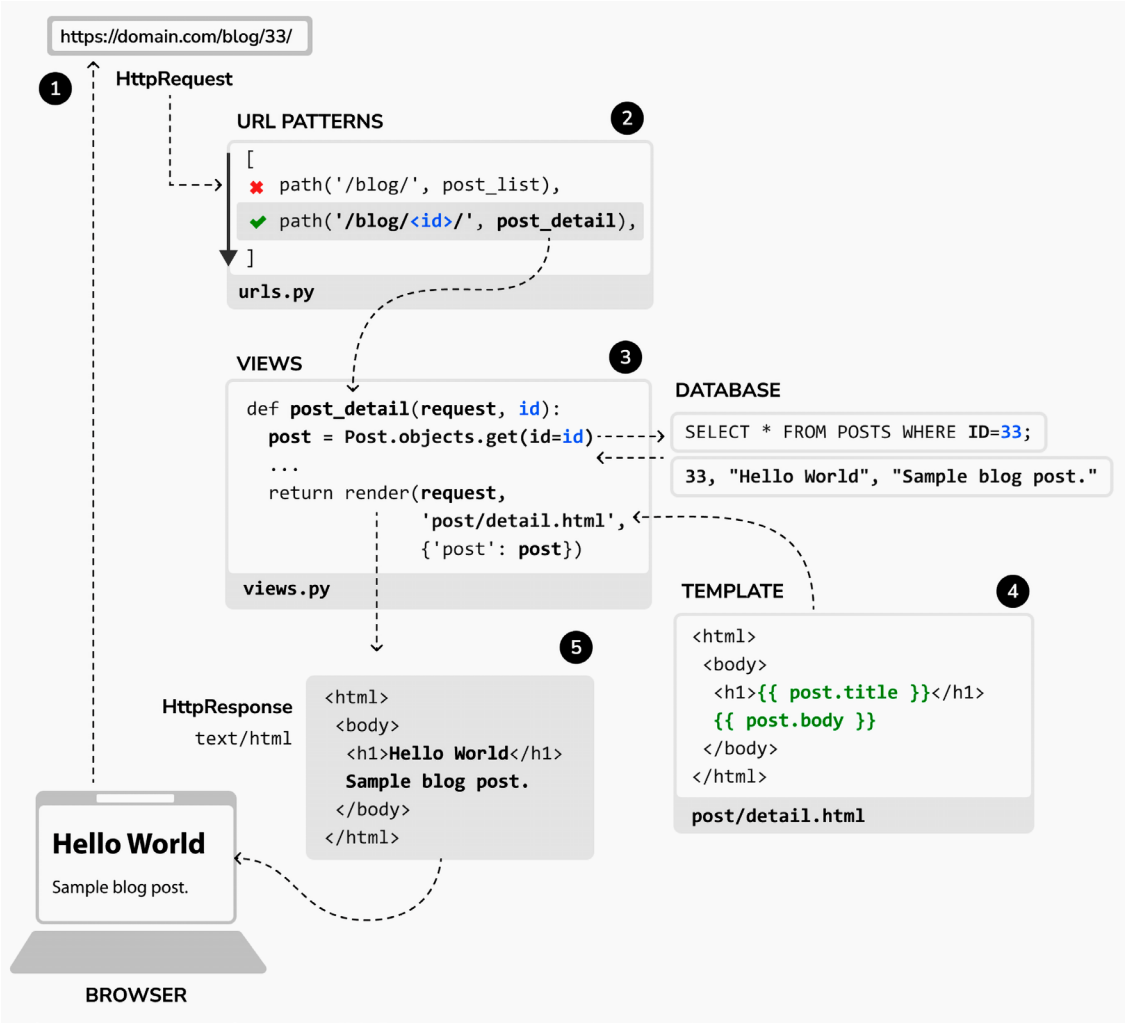


Figure 1.16: The Django request/response cycle

Let's review the Django request/response process:

1. A web browser requests a page by its URL, for example, `https://domain.com/blog/33/`. The web server receives the HTTP request and passes it over to Django.
2. Django runs through each URL pattern defined in the URL patterns configuration. The framework checks each pattern against the given URL path, in order of appearance, and stops at the first one that matches the requested URL. In this case, the pattern `/blog/<id>/` matches the path `/blog/33/`.

3. Django imports the view of the matching URL pattern and executes it, passing an instance of the `HttpRequest` class and the keyword or positional arguments. The view uses the models to retrieve information from the database. Using the Django ORM QuerySets are translated into SQL and executed in the database.
4. The view uses the `render()` function to render an HTML template passing the `Post` object as a context variable.
5. The rendered content is returned as a `HttpResponse` object by the view with the `text/html` content type by default.

You can always use this schema as the basic reference for how Django processes requests. This schema doesn't include Django middleware for the sake of simplicity. You will use middleware in different examples of this book, and you will learn how to create custom middleware in *Chapter 17*, *Going Live*.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter01`
- Python `venv` library for virtual environments – `https://docs.python.org/3/library/venv.html`
- Django installation options – `https://docs.djangoproject.com/en/4.1/topics/install/`
- Django 4.0 release notes – `https://docs.djangoproject.com/en/dev/releases/4.0/`
- Django 4.1 release notes – `https://docs.djangoproject.com/en/4.1/releases/4.1/`
- Django's design philosophies – `https://docs.djangoproject.com/en/dev/misc/design-philosophies/`
- Django model field reference – `https://docs.djangoproject.com/en/4.1/ref/models/fields/`
- Model index reference – `https://docs.djangoproject.com/en/4.1/ref/models/indexes/`
- Python support for enumerations – `https://docs.python.org/3/library/enum.html`
- Django model enumeration types – `https://docs.djangoproject.com/en/4.1/ref/models/fields/#enumeration-types`
- Django settings reference – `https://docs.djangoproject.com/en/4.1/ref/settings/`
- Django administration site – `https://docs.djangoproject.com/en/4.1/ref/contrib/admin/`
- Making queries with the Django ORM – `https://docs.djangoproject.com/en/4.1/topics/db/queries/`
- Django URL dispatcher – `https://docs.djangoproject.com/en/4.1/topics/http/urls/`
- Django URL resolver utilities – `https://docs.djangoproject.com/en/4.1/ref/urlresolvers/`
- Django template language – `https://docs.djangoproject.com/en/4.1/ref/templates/language/`

- Built-in template tags and filters – `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/`
- Static files for the code in this chapter – `https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter01/mysite/blog/static`

Summary

In this chapter, you learned the basics of the Django web framework by creating a simple blog application. You designed the data models and applied migrations to the database. You also created the views, templates, and URLs for your blog.

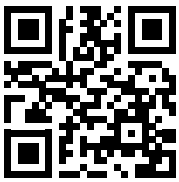
In the next chapter, you will learn how to create canonical URLs for models and how to build SEO-friendly URLs for blog posts. You will also learn how to implement object pagination and how to build class-based views. You will also implement Django forms to let your users recommend posts by email and comment on posts.

Join us on Discord

Read this book alongside other users and the author.

Ask questions, provide solutions to other readers, chat with the author via *Ask Me Anything* sessions, and much more. Scan the QR code or visit the link to join the book community.

`https://packt.link/django`



2

Enhancing Your Blog with Advanced Features

In the preceding chapter, we learned the main components of Django by developing a simple blog application. We created a simple blog application using views, templates, and URLs. In this chapter, we will extend the functionalities of the blog application with features that can be found in many blogging platforms nowadays. In this chapter, you will learn the following topics:

- Using canonical URLs for models
- Creating SEO-friendly URLs for posts
- Adding pagination to the post list view
- Building class-based views
- Sending emails with Django
- Using Django forms to share posts via email
- Adding comments to posts using forms from models

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02`.

All Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all the requirements at once with the command `pip install -r requirements.txt`.

Using canonical URLs for models

A website might have different pages that display the same content. In our application, the initial part of the content for each post is displayed both on the post list page and the post detail page. A canonical URL is the preferred URL for a resource. You can think of it as the URL of the most representative page for specific content. There might be different pages on your site that display posts, but there is a single URL that you use as the main URL for a post. Canonical URLs allow you to specify the URL for the master copy of a page. Django allows you to implement the `get_absolute_url()` method in your models to return the canonical URL for the object.

We will use the `post_detail` URL defined in the URL patterns of the application to build the canonical URL for `Post` objects. Django provides different URL resolver functions that allow you to build URLs dynamically using their name and any required parameters. We will use the `reverse()` utility function of the `django.urls` module.

Edit the `models.py` file of the `blog` application to import the `reverse()` function and add the `get_absolute_url()` method to the `Post` model as follows. New code is highlighted in bold:

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
from django.urls import reverse

class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
                      .filter(status=Post.Status.PUBLISHED)

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]
```

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

    def get_absolute_url(self):
        return reverse('blog:post_detail',
                       args=[self.id])
```

The `reverse()` function will build the URL dynamically using the URL name defined in the URL patterns. We have used the `blog` namespace followed by a colon and the URL name `post_detail`. Remember that the `blog` namespace is defined in the main `urls.py` file of the project when including the URL patterns from `blog.urls`. The `post_detail` URL is defined in the `urls.py` file of the `blog` application. The resulting string, `blog:post_detail`, can be used globally in your project to refer to the post detail URL. This URL has a required parameter that is the `id` of the blog post to retrieve. We have included the `id` of the `Post` object as a positional argument by using `args=[self.id]`.

You can learn more about the URL's utility functions at `https://docs.djangoproject.com/en/4.1/ref/urlresolvers/`.

Let's replace the post detail URLs in the templates with the new `get_absolute_url()` method.

Edit the `blog/post/list.html` file and replace the line:

```
<a href="{% url 'blog:post_detail' post.id %}">
```

With the line:

```
<a href="{{ post.get_absolute_url }}">
```

The `blog/post/list.html` file should now look as follows:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
```

```
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

Open the shell prompt and execute the following command to start the development server:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/blog/` in your browser. Links to individual blog posts should still work. Django is now building them using the `get_absolute_url()` method of the `Post` model.

Creating SEO-friendly URLs for posts

The canonical URL for a blog post detail view currently looks like `/blog/1/`. We will change the URL pattern to create SEO-friendly URLs for posts. We will be using both the `publish` date and `slug` values to build the URLs for single posts. By combining dates, we will make a post detail URL to look like `/blog/2022/1/1/who-was-django-reinhardt/`. We will provide search engines with friendly URLs to index, containing both the title and date of the post.

To retrieve single posts with the combination of publication date and slug, we need to ensure that no post can be stored in the database with the same `slug` and `publish` date as an existing post. We will prevent the `Post` model from storing duplicated posts by defining slugs to be unique for the publication date of the post.

Edit the `models.py` file and add the following `unique_for_date` parameter to the `slug` field of the `Post` model:

```
class Post(models.Model):
    # ...
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    # ...
```

By using `unique_for_date`, the `slug` field is now required to be unique for the date stored in the `publish` field. Note that the `publish` field is an instance of `DateTimeField`, but the check for unique values will be done only against the date (not the time). Django will prevent from saving a new post with the same slug as an existing post for a given publication date. We have now ensured that slugs are unique for the publication date, so we can now retrieve single posts by the `publish` and `slug` fields.

We have changed our models, so let's create migrations. Note that `unique_for_date` is not enforced at the database level, so no database migration is required. However, Django uses migrations to keep track of all model changes. We will create a migration just to keep migrations aligned with the current state of the model.

Run the following command in the shell prompt:

```
python manage.py makemigrations blog
```

You should get the following output:

```
Migrations for 'blog':
    blog/migrations/0002_alter_post_slug.py
    - Alter field slug on post
```

Django just created the `0002_alter_post_slug.py` file inside the `migrations` directory of the `blog` application.

Execute the following command in the shell prompt to apply existing migrations:

```
python manage.py migrate
```

You will get an output that ends with the following line:

```
Applying blog.0002_alter_post_slug... OK
```

Django will consider that all migrations have been applied and the models are in sync. No action will be done in the database because `unique_for_date` is not enforced at the database level.

Modifying the URL patterns

Let's modify the URL patterns to use the publication date and slug for the post detail URL.

Edit the `urls.py` file of the `blog` application and replace the line:

```
path('<int:id>/', views.post_detail, name='post_detail'),
```

With the lines:

```
path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
```

The `urls.py` file should now look like this:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
]
```

The URL pattern for the `post_detail` view takes the following arguments:

- `year`: Requires an integer
- `month`: Requires an integer
- `day`: Requires an integer
- `post`: Requires a slug (a string that contains only letters, numbers, underscores, or hyphens)

The `int` path converter is used for the `year`, `month`, and `day` parameters, whereas the `slug` path converter is used for the `post` parameter. You learned about path converters in the previous chapter. You can see all path converters provided by Django at `https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters`.

Modifying the views

Now we have to change the parameters of the `post_detail` view to match the new URL parameters and use them to retrieve the corresponding `Post` object.

Edit the `views.py` file and edit the `post_detail` view like this:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

We have modified the `post_detail` view to take the `year`, `month`, `day`, and `post` arguments and retrieve a published post with the given slug and publication date. By adding `unique_for_date='publish'` to the `slug` field of the `Post` model before, we ensured that there will be only one post with a slug for a given date. Thus, you can retrieve single posts using the date and slug.

Modifying the canonical URL for posts

We also have to modify the parameters of the canonical URL for blog posts to match the new URL parameters.

Edit the `models.py` file of the `blog` application and edit the `get_absolute_url()` method as follows:

```
class Post(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('blog:post_detail',
```

```
                    args=[self.publish.year,
                          self.publish.month,
                          self.publish.day,
                          self.slug])
```

Start the development server by typing the following command in the shell prompt:

```
python manage.py runserver
```

Next, you can return to your browser and click on one of the post titles to take a look at the detail view of the post. You should see something like this:

Figure 2.1: The page for the post's detail view

Take a look at the URL—it should look like `/blog/2022/1/1/who-was-django-reinhardt/`. You have designed SEO-friendly URLs for the blog posts.

Adding pagination

When you start adding content to your blog, you can easily store tens or hundreds of posts in your database. Instead of displaying all the posts on a single page, you may want to split the list of posts across several pages and include navigation links to the different pages. This functionality is called pagination, and you can find it in almost every web application that displays long lists of items.

For example, Google uses pagination to divide search results across multiple pages. *Figure 2.2* shows Google's pagination links for search result pages:

Figure 2.2: Google pagination links for search result pages

Django has a built-in pagination class that allows you to manage paginated data easily. You can define the number of objects you want to be returned per page and you can retrieve the posts that correspond to the page requested by the user.

Adding pagination to the post list view

Edit the `views.py` file of the `blog` application to import the Django `Paginator` class and modify the `post_list` view as follows:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator

def post_list(request):
    post_list = Post.published.all()
    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    posts = paginator.page(page_number)

    return render(request,
                  'blog/post/list.html',
                  {'posts': posts})
```

Let's review the new code we have added to the view:

1. We instantiate the `Paginator` class with the number of objects to return per page. We will display three posts per page.
2. We retrieve the page `GET` HTTP parameter and store it in the `page_number` variable. This parameter contains the requested page number. If the `page` parameter is not in the `GET` parameters of the request, we use the default value `1` to load the first page of results.
3. We obtain the objects for the desired page by calling the `page()` method of `Paginator`. This method returns a `Page` object that we store in the `posts` variable.
4. We pass the page number and the `posts` object to the template.

Creating a pagination template

We need to create a page navigation for users to browse through the different pages. We will create a template to display the pagination links. We will make it generic so that we can reuse the template for any object pagination on our website.

In the `templates/` directory, create a new file and name it `pagination.html`. Add the following HTML code to the file:

```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
      <a href="?page={{ page.previous_page_number }}">Previous</a>
    {% endif %}
```

```
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}.
    </span>
    {% if page.has_next %}
      <a href="?page={{ page.next_page_number }}">Next</a>
    {% endif %}
  </span>
</div>
```

This is the generic pagination template. The template expects to have a `Page` object in the context to render the previous and next links, and to display the current page and total pages of results.

Let's return to the `blog/post/list.html` template and include the `pagination.html` template at the bottom of the `{% content %}` block, as follows:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

The `{% include %}` template tag loads the given template and renders it using the current template context. We use `with` to pass additional context variables to the template. The pagination template uses the `page` variable to render, while the `Page` object that we pass from our view to the template is called `posts`. We use `with page=posts` to pass the variable expected by the pagination template. You can follow this method to use the pagination template for any type of object.

Start the development server by typing the following command in the shell prompt:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/blog/post/` in your browser and use the administration site to create a total of four different posts. Make sure to set the status to **Published** for all of them.

Now, open `http://127.0.0.1:8000/blog/` in your browser. You should see the first three posts in reverse chronological order, and then the navigation links at the bottom of the post list like this:

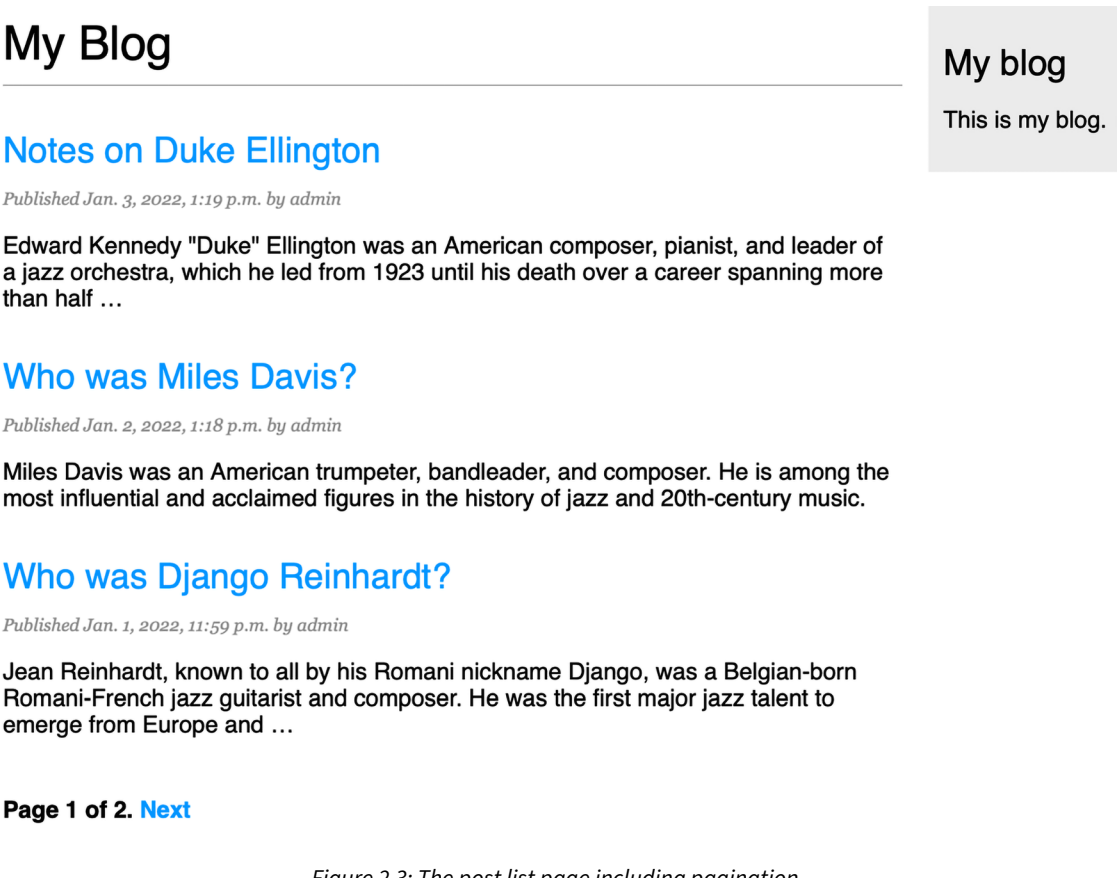


Figure 2.3: The post list page including pagination

If you click on **Next**, you will see the last post. The URL for the second page contains the `?page=2 GET` parameter. This parameter is used by the view to load the requested page of results using the paginator.

Figure 2.4: The second page of results

Great! The pagination links are working as expected.

Handling pagination errors

Now that the pagination is working, we can add exception handling for pagination errors in the view. The page parameter used by the view to retrieve the given page could potentially be used with wrong values, such as non-existing page numbers or a string value that cannot be used as a page number. We will implement appropriate error handling for those cases.

Open `http://127.0.0.1:8000/blog/?page=3` in your browser. You should see the following error page:

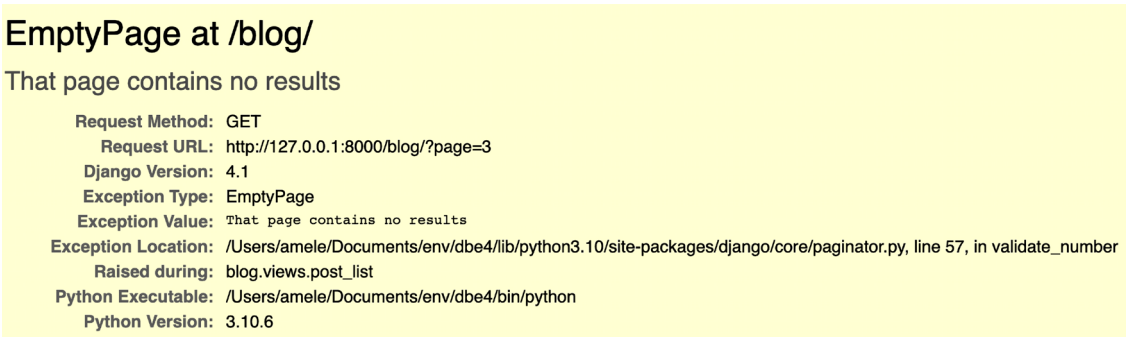


Figure 2.5: The EmptyPage error page

The `Paginator` object throws an `EmptyPage` exception when retrieving page 3 because it's out of range. There are no results to display. Let's handle this error in our view.

Edit the `views.py` file of the `blog` application to add the necessary imports and modify the `post_list` view as follows:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator, EmptyPage

def post_list(request):
    post_list = Post.published.all()
    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except EmptyPage:
        # If page_number is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                   {'posts': posts})
```

We have added a try and except block to manage the `EmptyPage` exception when retrieving a page. If the page requested is out of range, we return the last page of results. We get the total number of pages with `paginator.num_pages`. The total number of pages is the same as the last page number.

Open `http://127.0.0.1:8000/blog/?page=3` in your browser again. Now, the exception is managed by the view and the last page of results is returned as follows:

Figure 2.6: The last page of results

Our view should also handle the case when something different than an integer is passed in the page parameter.

Open `http://127.0.0.1:8000/blog/?page=asdf` in your browser. You should see the following error page:

Figure 2.7: The PageNotAnInteger error page

In this case, the `Paginator` object throws a `PageNotAnInteger` exception when retrieving the page `asdf` because page numbers can only be an integer. Let's handle this error in our view.

Edit the `views.py` file of the `blog` application to add the necessary imports and modify the `post_list` view as follows:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator, EmptyPage,\
                                  PageNotAnInteger

def post_list(request):
    post_list = Post.published.all()
    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page')
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # If page_number is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
```

```
        # If page_number is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                   {'posts': posts})
```

We have added a new except block to manage the `PageNotAnInteger` exception when retrieving a page. If the page requested is not an integer, we return the first page of results.

Open `http://127.0.0.1:8000/blog/?page=asdf` in your browser again. Now the exception is managed by the view and the first page of results is returned as follows:

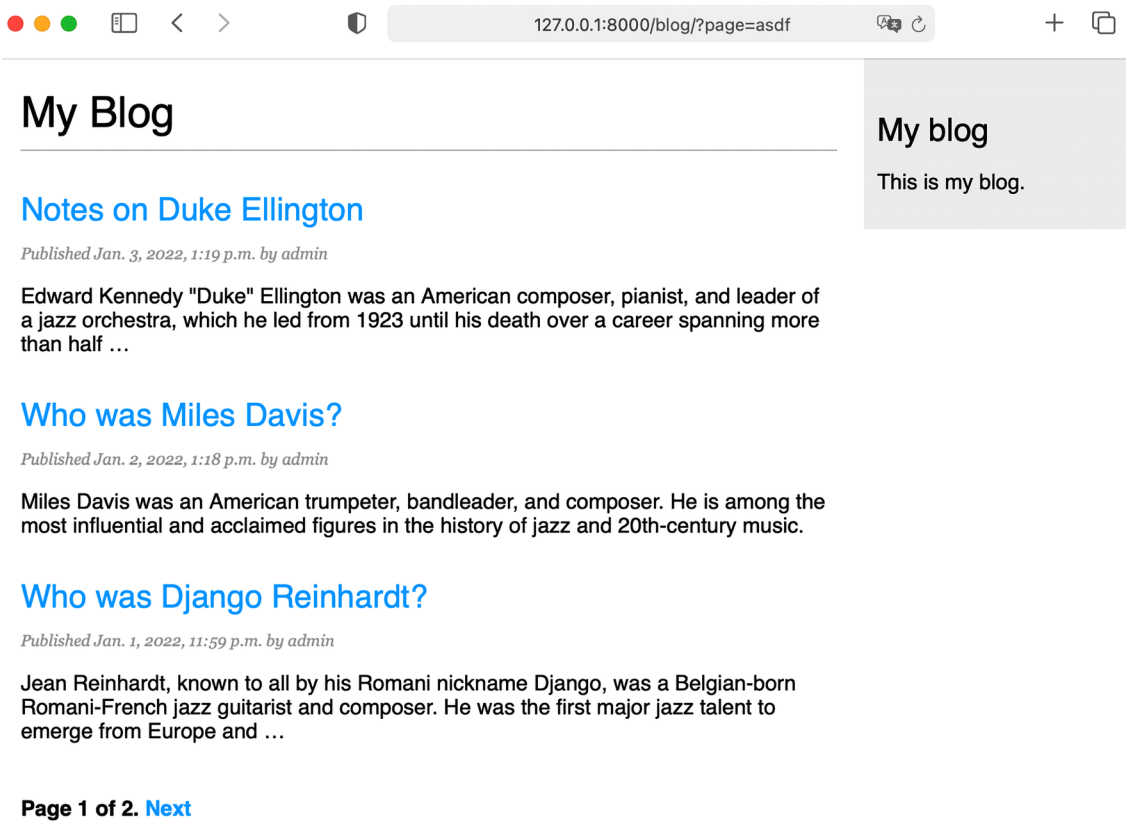


Figure 2.8: The first page of results

The pagination for blog posts is now fully implemented.

You can learn more about the `Paginator` class at `https://docs.djangoproject.com/en/4.1/ref/paginator/`.

Building class-based views

We have built the blog application using function-based views. Function-based views are simple and powerful, but Django also allows you to build views using classes.

Class-based views are an alternative way to implement views as Python objects instead of functions. Since a view is a function that takes a web request and returns a web response, you can also define your views as class methods. Django provides base view classes that you can use to implement your own views. All of them inherit from the `View` class, which handles HTTP method dispatching and other common functionalities.

Why use class-based views

Class-based views offer some advantages over function-based views that are useful for specific use cases. Class-based views allow you to:

- Organize code related to HTTP methods, such as `GET`, `POST`, or `PUT`, in separate methods, instead of using conditional branching
- Use multiple inheritance to create reusable view classes (also known as *mixins*)

Using a class-based view to list posts

To understand how to write class-based views, we will create a new class-based view that is equivalent to the `post_list` view. We will create a class that will inherit from the generic `ListView` view offered by Django. `ListView` allows you to list any type of object.

Edit the `views.py` file of the `blog` application and add the following code to it:

```
from django.views.generic import ListView

class PostListView(ListView):
    """
    Alternative post list view
    """
    queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'
```

The `PostListView` view is analogous to the `post_list` view we built previously. We have implemented a class-based view that inherits from the `ListView` class. We have defined a view with the following attributes:

- We use `queryset` to use a custom QuerySet instead of retrieving all objects. Instead of defining a `queryset` attribute, we could have specified `model = Post` and Django would have built the generic `Post.objects.all()` QuerySet for us.

- We use the context variable `posts` for the query results. The default variable is `object_list` if you don't specify any `context_object_name`.
- We define the pagination of results with `paginate_by`, returning three objects per page.
- We use a custom template to render the page with `template_name`. If you don't set a default template, `ListView` will use `blog/post_list.html` by default.

Now, edit the `urls.py` file of the `blog` application, comment the preceding `post_list` URL pattern, and add a new URL pattern using the `PostListView` class, as follows:

```
urlpatterns = [
    # Post views
    # path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
        views.post_detail,
        name='post_detail'),
]
```

In order to keep pagination working, we have to use the right page object that is passed to the template. Django's `ListView` generic view passes the page requested in a variable called `page_obj`. We have to edit the `post/list.html` template accordingly to include the paginator using the right variable, as follows:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=page_obj %}
{% endblock %}
```

Open `http://127.0.0.1:8000/blog/` in your browser and verify that the pagination links work as expected. The behavior of the pagination links should be the same as with the previous `post_list` view.

The exception handling in this case is a bit different. If you try to load a page out of range or pass a non-integer value in the `page` parameter, the view will return an HTTP response with the status code `404` (page not found) like this:

Figure 2.9: HTTP 404 Page not found response

The exception handling that returns the HTTP `404` status code is provided by the `ListView` view.

This is a simple example of how to write class-based views. You will learn more about class-based views in *Chapter 13, Creating a Content Management System*, and successive chapters.

You can read an introduction to class-based views at `https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/`.

Recommending posts by email

Now, we will learn how to create forms and how to send emails with Django. We will allow users to share blog posts with others by sending post recommendations via email.

Take a minute to think about how you could use *views*, *URLs*, and *templates* to create this functionality using what you learned in the preceding chapter.

To allow users to share posts via email, we will need to:

- Create a form for users to fill in their name, their email address, the recipient email address, and optional comments
- Create a view in the `views.py` file that handles the posted data and sends the email
- Add a URL pattern for the new view in the `urls.py` file of the blog application
- Create a template to display the form

Creating forms with Django

Let's start by building the form to share posts. Django has a built-in forms framework that allows you to create forms easily. The forms framework makes it simple to define the fields of the form, specify how they have to be displayed, and indicate how they have to validate input data. The Django forms framework offers a flexible way to render forms in HTML and handle data.

Django comes with two base classes to build forms:

- `Form`: Allows you to build standard forms by defining fields and validations.
- `ModelForm`: Allows you to build forms tied to model instances. It provides all the functionalities of the base `Form` class, but form fields can be explicitly declared, or automatically generated, from model fields. The form can be used to create or edit model instances.

First, create a `forms.py` file inside the directory of your `blog` application and add the following code to it:

```
from django import forms

class EmailPostForm(forms.Form):
    name = forms.CharField(max_length=25)
    email = forms.EmailField()
    to = forms.EmailField()
    comments = forms.CharField(required=False,
                               widget=forms.Textarea)
```

We have defined our first Django form. The `EmailPostForm` form inherits from the base `Form` class. We use different field types to validate data accordingly.

Forms can reside anywhere in your Django project. The convention is to place them inside a `forms.py` file for each application.

The form contains the following fields:

- `name`: An instance of `CharField` with a maximum length of `25` characters. We will use it for the name of the person sending the post.
- `email`: An instance of `EmailField`. We will use the email of the person sending the post recommendation.
- `to`: An instance of `EmailField`. We will use the email of the recipient, who will receive the email recommending the post recommendation.
- `comments`: An instance of `CharField`. We will use it for comments to include in the post recommendation email. We have made this field optional by setting `required` to `False`, and we have specified a custom widget to render the field.

Each field type has a default widget that determines how the field is rendered in HTML. The `name` field is an instance of `CharField`. This type of field is rendered as an `<input type="text">` HTML element. The default widget can be overridden with the `widget` attribute. In the `comments` field, we use the `Textarea` widget to display it as a `<textarea>` HTML element instead of the default `<input>` element.

Field validation also depends on the field type. For example, the `email` and `to` fields are `EmailField` fields. Both fields require a valid email address; the field validation will otherwise raise a `forms.ValidationError` exception and the form will not validate. Other parameters are also taken into account for the form field validation, such as the `name` field having a maximum length of `25` or the `comments` field being optional.

These are only some of the field types that Django provides for forms. You can find a list of all field types available at `https://docs.djangoproject.com/en/4.1/ref/forms/fields/`.

Handling forms in views

We have defined the form to recommend posts via email. Now we need a view to create an instance of the form and handle the form submission.

Edit the `views.py` file of the `blog` application and add the following code to it:

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # Retrieve post by id
    post = get_object_or_404(Post, id=post_id, status=Post.Status.PUBLISHED)
    if request.method == 'POST':
        # Form was submitted
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Form fields passed validation
            cd = form.cleaned_data
            # ... send email
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form})
```

We have defined the `post_share` view that takes the `request` object and the `post_id` variable as parameters. We use the `get_object_or_404()` shortcut to retrieve a published post by its `id`.

We use the same view both for displaying the initial form and processing the submitted data. The HTTP `request` method allows us to differentiate whether the form is being submitted. A `GET` request will indicate that an empty form has to be displayed to the user and a `POST` request will indicate the form is being submitted. We use `request.method == 'POST'` to differentiate between the two scenarios.

This is the process to display the form and handle the form submission:

1. When the page is loaded for the first time, the view receives a `GET` request. In this case, a new `EmailPostForm` instance is created and stored in the `form` variable. This form instance will be used to display the empty form in the template:

```
form = EmailPostForm()
```

2. When the user fills in the form and submits it via `POST`, a form instance is created using the submitted data contained in `request.POST`:

```
if request.method == 'POST':
    # Form was submitted
    form = EmailPostForm(request.POST)
```

3. After this, the data submitted is validated using the form's `is_valid()` method. This method validates the data introduced in the form and returns `True` if all fields contain valid data. If any field contains invalid data, then `is_valid()` returns `False`. The list of validation errors can be obtained with `form.errors`.
4. If the form is not valid, the form is rendered in the template again, including the data submitted. Validation errors will be displayed in the template.
5. If the form is valid, the validated data is retrieved with `form.cleaned_data`. This attribute is a dictionary of form fields and their values.

If your form data does not validate, `cleaned_data` will contain only the valid fields.

We have implemented the view to display the form and handle the form submission. We will now learn how to send emails using Django and then we will add that functionality to the `post_share` view.

Sending emails with Django

Sending emails with Django is very straightforward. To send emails with Django, you need to have a local **Simple Mail Transfer Protocol** (**SMTP**) server, or you need to access an external SMTP server, like your email service provider.

The following settings allow you to define the SMTP configuration to send emails with Django:

- `EMAIL_HOST`: The SMTP server host; the default is `localhost`
- `EMAIL_PORT`: The SMTP port; the default is `25`
- `EMAIL_HOST_USER`: The username for the SMTP server
- `EMAIL_HOST_PASSWORD`: The password for the SMTP server

- `EMAIL_USE_TLS`: Whether to use a **Transport Layer Security** (**TLS**) secure connection
- `EMAIL_USE_SSL`: Whether to use an implicit TLS secure connection

For this example, we will use Google's SMTP server with a standard Gmail account.

If you have a Gmail account, edit the `settings.py` file of your project and add the following code to it:

```
# Email server configuration
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = ''
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Replace `your_account@gmail.com` with your actual Gmail account. If you don't have a Gmail account, you can use the SMTP server configuration of your email service provider.

Instead of Gmail, you can also use a professional, scalable email service that allows you to send emails via SMTP using your own domain, such as SendGrid (`https://sendgrid.com/`) or Amazon Simple Email Service (`https://aws.amazon.com/ses/`). Both services will require you to verify your domain and sender email accounts and will provide you with SMTP credentials to send emails. The Django applications `django-sengrid` and `django-ses` simplify the task of adding SendGrid or Amazon SES to your project. You can find installation instructions for `django-sengrid` at `https://github.com/sklarsa/django-sendgrid-v5`, and installation instructions for `django-ses` at `https://github.com/django-ses/django-ses`.

If you can't use an SMTP server, you can tell Django to write emails to the console by adding the following setting to the `settings.py` file:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

By using this setting, Django will output all emails to the shell instead of sending them. This is very useful for testing your application without an SMTP server.

To complete the Gmail configuration, we need to enter a password for the SMTP server. Since Google uses a two-step verification process and additional security measures, you cannot use your Google account password directly. Instead, Google allows you to create app-specific passwords for your account. An app password is a 16-digit passcode that gives a less secure app or device permission to access your Google account.

Open `https://myaccount.google.com/` in your browser. On the left menu, click on **Security**. You will see the following screen:

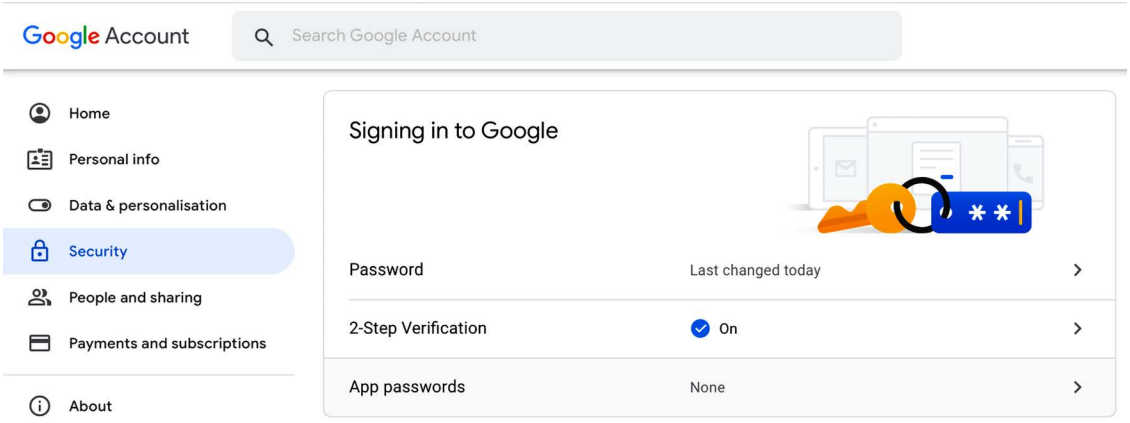


Figure 2.10: The Signing in to Google page for Google accounts

Under the **Signing in to Google** block, click on **App passwords**. If you cannot see **App passwords**, it might be that 2-step verification is not set for your account, your account is an organization account instead of a standard Gmail account, or you turned on Google's advanced protection. Make sure to use a standard Gmail account and to activate 2-step verification for your Google account. You can find more information at `https://support.google.com/accounts/answer/185833`.

When you click on **App passwords**, you will see the following screen:

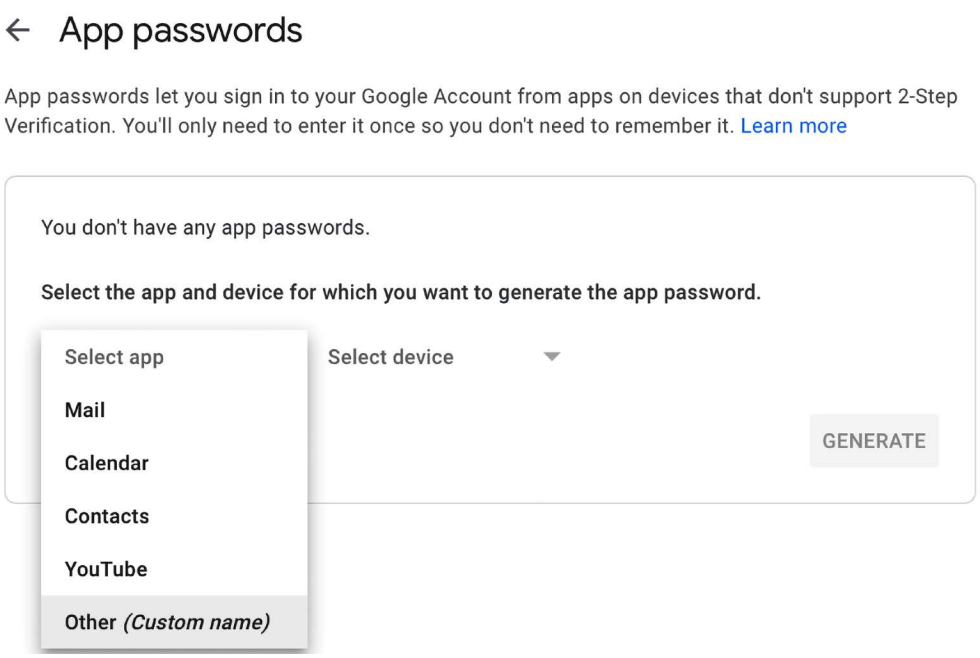


Figure 2.11: Form to generate a new Google app password

In the **Select app** dropdown, select **Other**.

Then, enter the name `Blog` and click the **GENERATE** button, as follows:

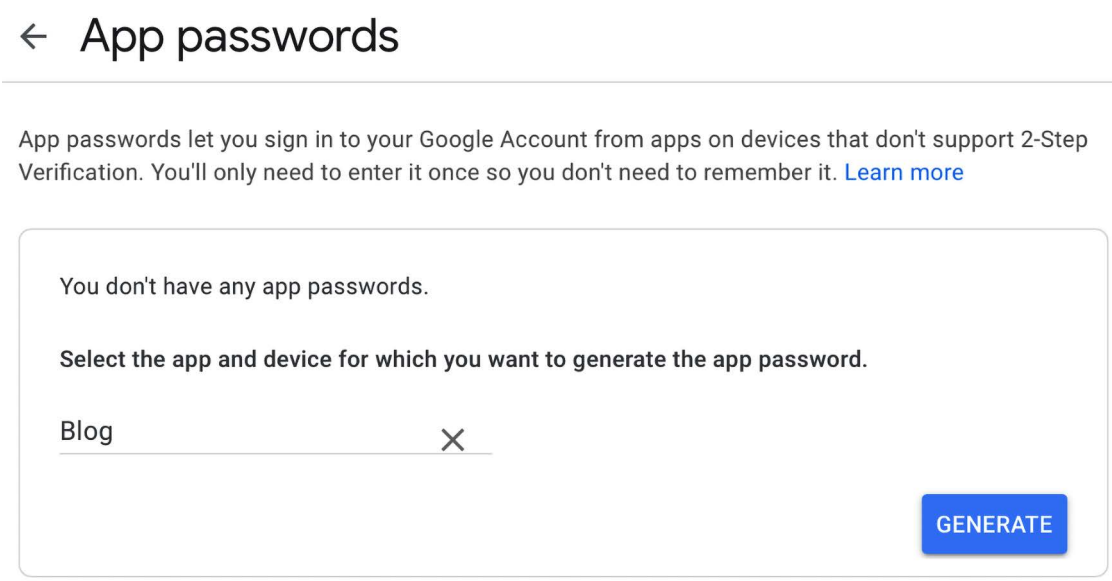


Figure 2.12: Form to generate a new Google app password

A new password will be generated and displayed to you like this:

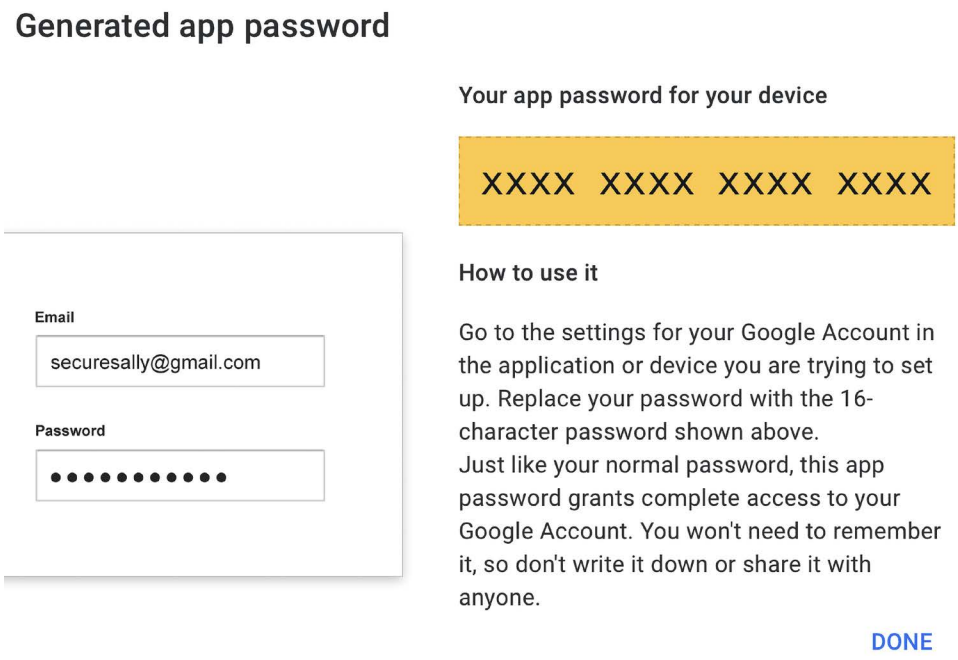


Figure 2.13: Generated Google app password

Copy the generated app password.

Edit the `settings.py` file of your project and add the app password to the `EMAIL_HOST_PASSWORD` setting, as follows:

```
# Email server configuration
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'xxxxxxxxxxxxxxxx'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Open the Python shell by running the following command in the system shell prompt:

```
python manage.py shell
```

Execute the following code in the Python shell:

```
>>> from django.core.mail import send_mail
>>> send_mail('Django mail',
...           'This e-mail was sent with Django.',
...           'your_account@gmail.com',
...           ['your_account@gmail.com'],
...           fail_silently=False)
```

The `send_mail()` function takes the subject, message, sender, and list of recipients as required arguments. By setting the optional argument `fail_silently=False`, we are telling it to raise an exception if the email cannot be sent. If the output you see is `1`, then your email was successfully sent.

Check your inbox. You should have received the email:

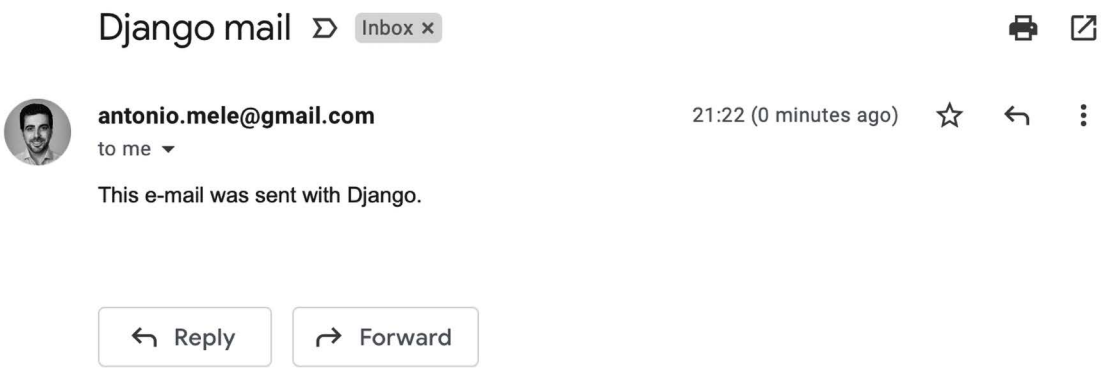


Figure 2.14: Test email sent displayed in Gmail

You just sent your first email with Django! You can find more information about sending emails with Django at `https://docs.djangoproject.com/en/4.1/topics/email/`.

Let's add this functionality to the `post_share` view.

Sending emails in views

Edit the `post_share` view in the `views.py` file of the `blog` application, as follows:

```
from django.core.mail import send_mail

def post_share(request, post_id):
    # Retrieve post by id
    post = get_object_or_404(Post, id=post_id, status=Post.Status.PUBLISHED)
    sent = False

    if request.method == 'POST':
        # Form was submitted
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Form fields passed validation
            cd = form.cleaned_data
            post_url = request.build_absolute_uri(
                post.get_absolute_url())
            subject = f"{cd['name']} recommends you read " \
                      f"{post.title}"
            message = f"Read {post.title} at {post_url}\n\n" \
                      f"{cd['name']}\'s comments: {cd['comments']}"
            send_mail(subject, message, 'your_account@gmail.com',
                      [cd['to']])
            sent = True
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form,
                                                    'sent': sent})
```

Replace `your_account@gmail.com` with your real email account if you are using an SMTP server instead of `console.EmailBackend`.

In the preceding code, we have declared a `sent` variable with the initial value `True`. We set this variable to `True` after the email is sent. We will use the `sent` variable later in the template to display a success message when the form is successfully submitted.

Since we have to include a link to the post in the email, we retrieve the absolute path of the post using its `get_absolute_url()` method. We use this path as an input for `request.build_absolute_uri()` to build a complete URL, including the HTTP schema and hostname.

We create the subject and the message body of the email using the cleaned data of the validated form. Finally, we send the email to the email address contained in the `to` field of the form.

Now that the view is complete, we have to add a new URL pattern for it.

Open the `urls.py` file of your `blog` application and add the `post_share` URL pattern, as follows:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    # path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
]
```

Rendering forms in templates

After creating the form, programming the view, and adding the URL pattern, the only thing missing is the template for the view.

Create a new file in the `blog/templates/blog/post/` directory and name it `share.html`.

Add the following code to the new `share.html` template:

```
{% extends "blog/base.html" %}

{% block title %}Share a post{% endblock %}

{% block content %}
  {% if sent %}
    <h1>E-mail successfully sent</h1>
    <p>
      "{{ post.title }}" was successfully sent to {{ form.cleaned_data.to }}.
    </p>
  {% else %}
    <h1>Share "{{ post.title }}" by e-mail</h1>
    <form method="post">
```

```
      {{ form.as_p }}
      {% csrf_token %}
      <input type="submit" value="Send e-mail">
    </form>
  {% endif %}
{% endblock %}
```

This is the template that is used to both display the form to share a post via email, and to display a success message when the email has been sent. We differentiate between both cases with `{% if sent %}`.

To display the form, we have defined an HTML form element, indicating that it has to be submitted by the `POST` method:

```
<form method="post">
```

We have included the form instance with `{{ form.as_p }}`. We tell Django to render the form fields using HTML paragraph `<p>` elements by using the `as_p` method. We could also render the form as an unordered list with `as_ul` or as an HTML table with `as_table`. Another option is to render each field by iterating through the form fields, as in the following example:

```
{% for field in form %}
  <div>
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
  </div>
{% endfor %}
```

We have added a `{% csrf_token %}` template tag. This tag introduces a hidden field with an autogenerated token to avoid **cross-site request forgery** (**CSRF**) attacks. These attacks consist of a malicious website or program performing an unwanted action for a user on the site. You can find more information about CSRF at `https://owasp.org/www-community/attacks/csrf`.

The `{% csrf_token %}` template tag generates a hidden field that is rendered like this:

```
<input type='hidden' name='csrfmiddlewaretoken'
value='26JjKo2lcEtYkGoV9z4XmJIEHLXN5LDR' />
```

By default, Django checks for the CSRF token in all `POST` requests. Remember to include the `csrf_token` tag in all forms that are submitted via `POST`.

Edit the `blog/post/detail.html` template and make it look like this:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}
```

```
{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
{% endblock %}
```

We have added a link to the `post_share` URL. The URL is built dynamically with the `{% url %}` template tag provided by Django. We use the namespace called `blog` and the URL named `post_share`. We pass the post `id` as a parameter to build the URL.

Open the shell prompt and execute the following command to start the development server:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/blog/` in your browser and click on any post title to view the post detail page.

Under the post body, you should see the link that you just added, as shown in *Figure 2.15*:

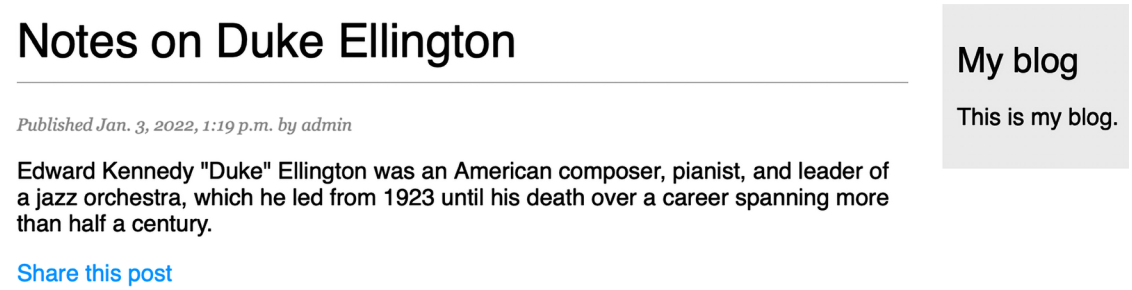


Figure 2.15: The post detail page, including a link to share the post

Click on **Share this post**, and you should see the page, including the form to share this post by email, as follows:

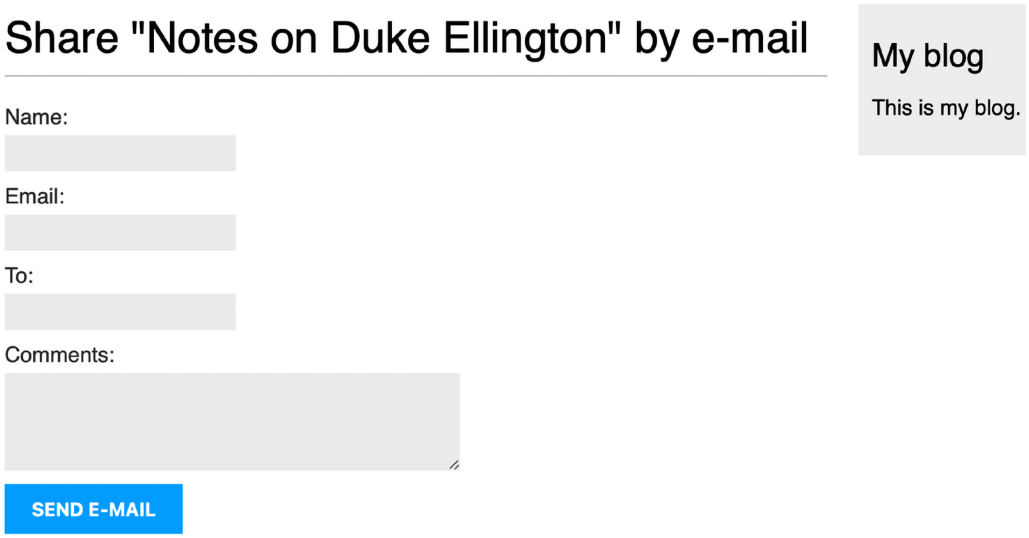


Figure 2.16: The page to share a post via email

CSS styles for the form are included in the example code in the `static/css/blog.css` file. When you click on the **SEND E-MAIL** button, the form is submitted and validated. If all fields contain valid data, you get a success message, as follows:

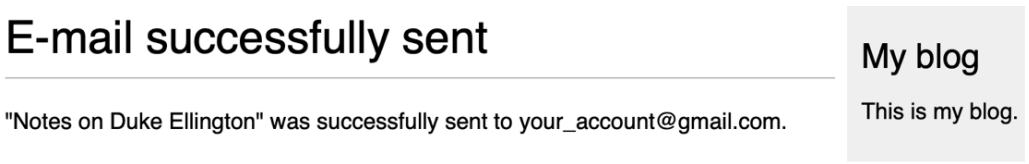


Figure 2.17: A success message for a post shared via email

Send a post to your own email address and check your inbox. The email you receive should look like this:

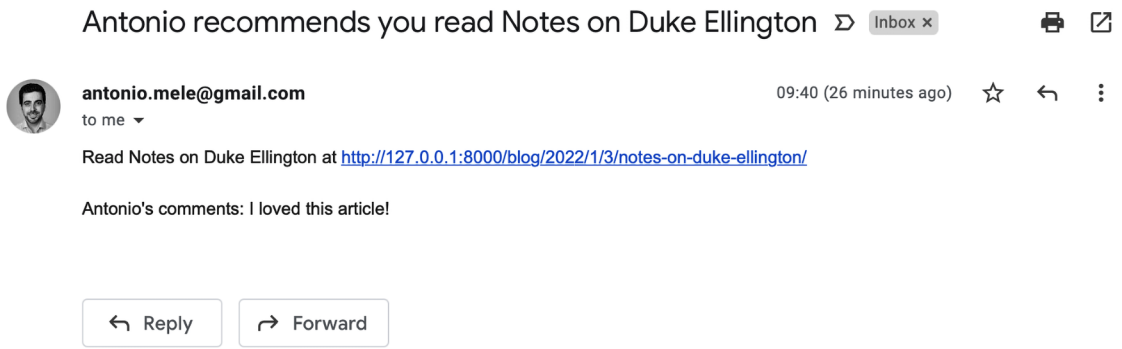


Figure 2.18: Test email sent displayed in Gmail

If you submit the form with invalid data, the form will be rendered again, including all validation errors:

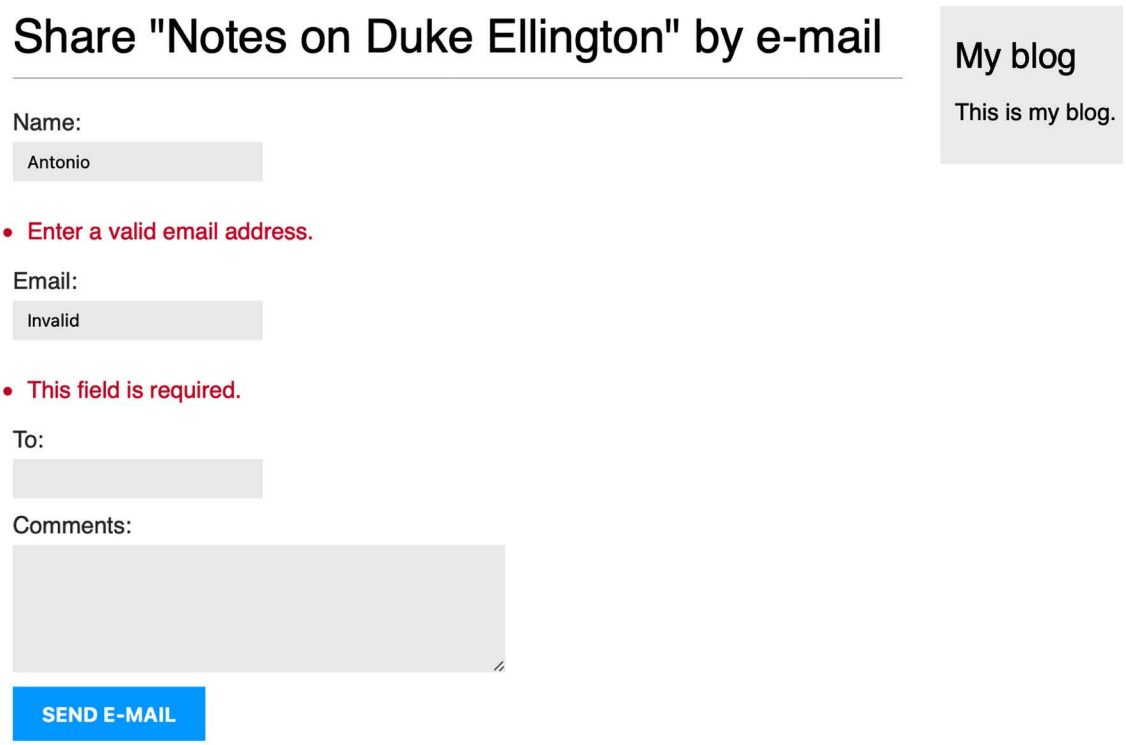


Figure 2.19: The share post form displaying invalid data errors

Most modern browsers will prevent you from submitting a form with empty or erroneous fields. This is because the browser validates the fields based on their attributes before submitting the form. In this case, the form won't be submitted, and the browser will display an error message for the fields that are wrong. To test the Django form validation using a modern browser, you can skip the browser form validation by adding the `novalidate` attribute to the HTML `<form>` element, like `<form method="post" novalidate>`. You can add this attribute to prevent the browser from validating fields and test your own form validation. After you are done testing, remove the `novalidate` attribute to keep the browser form validation.

The functionality for sharing posts by email is now complete. You can find more information about working with forms at `https://docs.djangoproject.com/en/4.1/topics/forms/`.

Creating a comment system

We will continue extending our blog application with a comment system that will allow users to comment on posts. To build the comment system, we will need the following:

- A comment model to store user comments on posts
- A form that allows users to submit comments and manages the data validation

- A view that processes the form and saves a new comment to the database
- A list of comments and a form to add a new comment that can be included in the post detail template

Creating a model for comments

Let's start by building a model to store user comments on posts.

Open the `models.py` file of your `blog` application and add the following code:

```
class Comment(models.Model):
    post = models.ForeignKey(Post,
                             on_delete=models.CASCADE,
                             related_name='comments')
    name = models.CharField(max_length=80)
    email = models.EmailField()
    body = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    active = models.BooleanField(default=True)

    class Meta:
        ordering = ['created']
        indexes = [
            models.Index(fields=['created']),
        ]

    def __str__(self):
        return f'Comment by {self.name} on {self.post}'
```

This is the `Comment` model. We have added a `ForeignKey` field to associate each comment with a single post. This many-to-one relationship is defined in the `Comment` model because each comment will be made on one post, and each post may have multiple comments.

The `related_name` attribute allows you to name the attribute that you use for the relationship from the related object back to this one. We can retrieve the post of a comment object using `comment.post` and retrieve all comments associated with a post object using `post.comments.all()`. If you don't define the `related_name` attribute, Django will use the name of the model in lowercase, followed by `_set` (that is, `comment_set`) to name the relationship of the related object to the object of the model, where this relationship has been defined.

You can learn more about many-to-one relationships at `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/`.

We have defined the `active` Boolean field to control the status of the comments. This field will allow us to manually deactivate inappropriate comments using the administration site. We use `default=True` to indicate that all comments are active by default.

We have defined the `created` field to store the date and time when the comment was created. By using `auto_now_add`, the date will be saved automatically when creating an object. In the `Meta` class of the model, we have added `ordering = ['created']` to sort comments in chronological order by default, and we have added an index for the `created` field in ascending order. This will improve the performance of database lookups or ordering results using the `created` field.

The `Comment` model that we have built is not synchronized into the database. We need to generate a new database migration to create the corresponding database table.

Run the following command from the shell prompt:

```
python manage.py makemigrations blog
```

You should see the following output:

```
Migrations for 'blog':
  blog/migrations/0003_comment.py
    - Create model Comment
```

Django has generated a `0003_comment.py` file inside the `migrations/` directory of the `blog` application. We need to create the related database schema and apply the changes to the database.

Run the following command to apply existing migrations:

```
python manage.py migrate
```

You will get an output that includes the following line:

```
Applying blog.0003_comment... OK
```

The migration has been applied and the `blog_comment` table has been created in the database.

Adding comments to the administration site

Next, we will add the new model to the administration site to manage comments through a simple interface.

Open the `admin.py` file of the `blog` application, import the `Comment` model, and add the following `ModelAdmin` class:

```
from .models import Post, Comment

@admin.register(Comment)
class CommentAdmin(admin.ModelAdmin):
    list_display = ['name', 'email', 'post', 'created', 'active']
    list_filter = ['active', 'created', 'updated']
    search_fields = ['name', 'email', 'body']
```

Open the shell prompt and execute the following command to start the development server:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/` in your browser. You should see the new model included in the **BLOG** section, as shown in *Figure 2.20*:

Figure 2.20: Blog application models on the Django administration index page

The model is now registered on the administration site.

In the **Comments** row, click on **Add**. You will see the form to add a new comment:

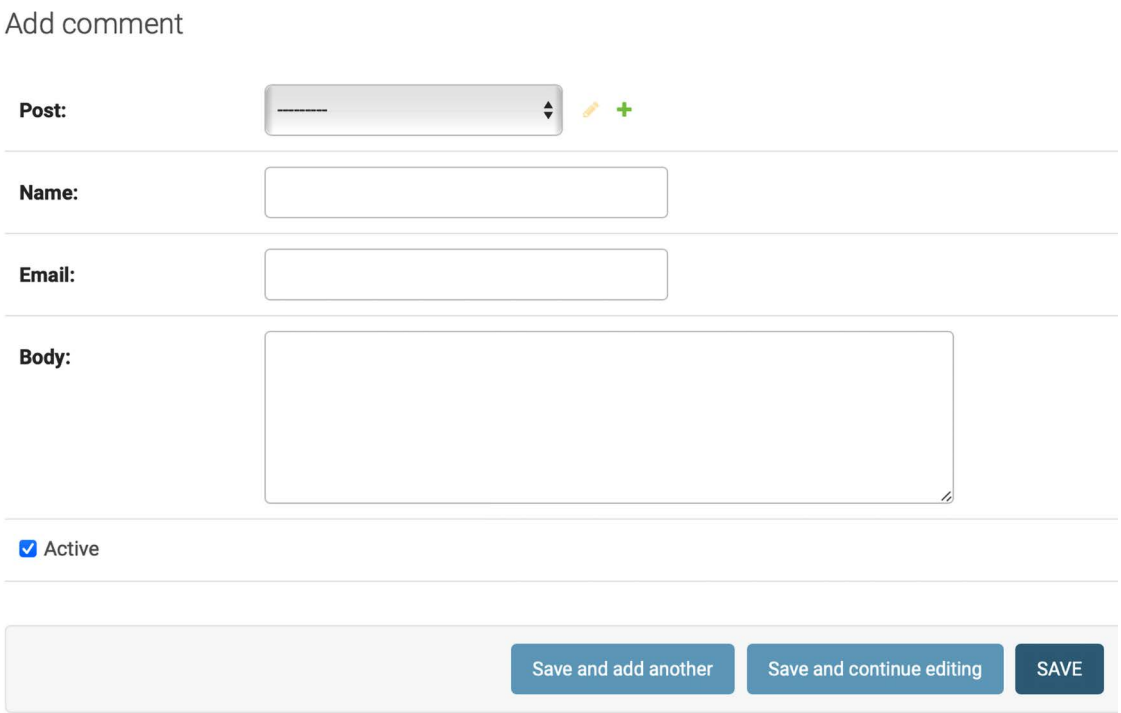


Figure 2.21: Blog application models on the Django administration index page

Now we can manage `Comment` instances using the administration site.

Creating forms from models

We need to build a form to let users comment on blog posts. Remember that Django has two base classes that can be used to create forms: `Form` and `ModelForm`. We used the `Form` class to allow users to share posts by email. Now we will use `ModelForm` to take advantage of the existing `Comment` model and build a form dynamically for it.

Edit the `forms.py` file of your `blog` application and add the following lines:

```
from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ['name', 'email', 'body']
```

To create a form from a model, we just indicate which model to build the form for in the `Meta` class of the form. Django will introspect the model and build the corresponding form dynamically.

Each model field type has a corresponding default form field type. The attributes of model fields are taken into account for form validation. By default, Django creates a form field for each field contained in the model. However, we can explicitly tell Django which fields to include in the form using the `fields` attribute or define which fields to exclude using the `exclude` attribute. In the `CommentForm` form, we have explicitly included the `name`, `email`, and `body` fields. These are the only fields that will be included in the form.

You can find more information about creating forms from models at `https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/`.

Handling ModelForms in views

For sharing posts by email, we used the same view to display the form and manage its submission. We used the HTTP method to differentiate between both cases; `GET` to display the form and `POST` to submit it. In this case, we will add the comment form to the post detail page, and we will build a separate view to handle the form submission. The new view that processes the form will allow the user to return to the post detail view once the comment has been stored in the database.

Edit the `views.py` file of the `blog` application and add the following code:

```
from django.shortcuts import render, get_object_or_404, redirect
from .models import Post, Comment
from django.core.paginator import Paginator, EmptyPage,\
                                  PageNotAnInteger
from django.views.generic import ListView
from .forms import EmailPostForm, CommentForm
from django.core.mail import send_mail
```

```
from django.views.decorators.http import require_POST

# ...

@require_POST
def post_comment(request, post_id):
    post = get_object_or_404(Post, id=post_id, status=Post.Status.PUBLISHED)
    comment = None
    # A comment was posted
    form = CommentForm(data=request.POST)
    if form.is_valid():
        # Create a Comment object without saving it to the database
        comment = form.save(commit=False)
        # Assign the post to the comment
        comment.post = post
        # Save the comment to the database
        comment.save()
    return render(request, 'blog/post/comment.html',
                           {'post': post,
                            'form': form,
                            'comment': comment})
```

We have defined the `post_comment` view that takes the `request` object and the `post_id` variable as parameters. We will be using this view to manage the post submission. We expect the form to be submitted using the HTTP `POST` method. We use the `require_POST` decorator provided by Django to only allow `POST` requests for this view. Django allows you to restrict the HTTP methods allowed for views. Django will throw an HTTP `405` (method not allowed) error if you try to access the view with any other HTTP method.

In this view, we have implemented the following actions:

1. We retrieve a published post by its `id` using the `get_object_or_404()` shortcut.
2. We define a `comment` variable with the initial value `None`. This variable will be used to store the comment object when it gets created.
3. We instantiate the form using the submitted `POST` data and validate it using the `is_valid()` method. If the form is invalid, the template is rendered with the validation errors.
4. If the form is valid, we create a new `Comment` object by calling the form's `save()` method and assign it to the `new_comment` variable, as follows:

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

5. The `save()` method creates an instance of the model that the form is linked to and saves it to the database. If you call it using `commit=False`, the model instance is created but not saved to the database. This allows us to modify the object before finally saving it.

The `save()` method is available for `ModelForm` but not for `Form` instances since they are not linked to any model.

6. We assign the post to the comment we created:

```
comment.post = post
```

7. We save the new comment to the database by calling its `save()` method:

```
comment.save()
```

8. We render the template `blog/post/comment.html`, passing the `post`, `form`, and `comment` objects in the template context. This template doesn't exist yet; we will create it later.

Let's create a URL pattern for this view.

Edit the `urls.py` file of the `blog` application and add the following URL pattern to it:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    # path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
]
```

We have implemented the view to manage the submission of comments and their corresponding URL. Let's create the necessary templates.

Creating templates for the comment form

We will create a template for the comment form that we will use in two places:

- In the post detail template associated with the `post_detail` view to let users publish comments
- In the post comment template associated with the `post_comment` view to display the form again if there are any form errors.

We will create the form template and use the `{% include %}` template tag to include it in the two other templates.

In the `templates/blog/post/` directory, create a new `includes/` directory. Add a new file inside this directory and name it `comment_form.html`.

The file structure should look as follows:

```
templates/
  blog/
    post/
      includes/
        comment_form.html
      detail.html
      list.html
      share.html
```

Edit the new `blog/post/includes/comment_form.html` template and add the following code:

```
<h2>Add a new comment</h2>
<form action="{% url "blog:post_comment" post.id %}" method="post">
  {{ form.as_p }}
  {% csrf_token %}
  <p><input type="submit" value="Add comment"></p>
</form>
```

In this template, we build the `action` URL of the HTML `<form>` element dynamically using the `{% url %}` template tag. We build the URL of the `post_comment` view that will process the form. We display the form rendered in paragraphs and we include `{% csrf_token %}` for CSRF protection because this form will be submitted with the `POST` method.

Create a new file in the `templates/blog/post/` directory of the `blog` application and name it `comment.html`.

The file structure should now look as follows:

```
templates/
  blog/
    post/
      includes/
        comment_form.html
      comment.html
      detail.html
      list.html
      share.html
```

Edit the new `blog/post/comment.html` template and add the following code:

```
{% extends "blog/base.html" %}

{% block title %}Add a comment{% endblock %}

{% block content %}
  {% if comment %}
    <h2>Your comment has been added.</h2>
    <p><a href="{{ post.get_absolute_url }}">Back to the post</a></p>
  {% else %}
    {% include "blog/post/includes/comment_form.html" %}
  {% endif %}
{% endblock %}
```

This is the template for the post comment view. In this view, we expect the form to be submitted via the `POST` method. The template covers two different scenarios:

- If the form data submitted is valid, the `comment` variable will contain the `comment` object that was created, and a success message will be displayed.
- If the form data submitted is not valid, the `comment` variable will be `None`. In this case, we will display the comment form. We use the `{% include %}` template tag to include the `comment_form.html` template that we have previously created.

Adding comments to the post detail view

Edit the `views.py` file of the `blog` application and edit the `post_detail` view as follows:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)
    # List of active comments for this post
    comments = post.comments.filter(active=True)
    # Form for users to comment
    form = CommentForm()
    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                   'comments': comments,
                   'form': form})
```

Let's review the code we have added to the `post_detail` view:

- We have added a QuerySet to retrieve all active comments for the post, as follows:

```
comments = post.comments.filter(active=True)
```

- This QuerySet is built using the `post` object. Instead of building a QuerySet for the `Comment` model directly, we leverage the `post` object to retrieve the related `Comment` objects. We use the `comments` manager for the related `Comment` objects that we previously defined in the `Comment` model, using the `related_name` attribute of the `ForeignKey` field to the `Post` model.
- We have also created an instance of the comment form with `form = CommentForm()`.

Adding comments to the post detail template

We need to edit the `blog/post/detail.html` template to implement the following:

- Display the total number of comments for a post
- Display the list of comments
- Display the form for users to add a new comment

We will start by adding the total number of comments for a post.

Edit the `blog/post/detail.html` template and change it as follows:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
{% endblock %}
```

We use the Django ORM in the template, executing the `comments.count()` QuerySet. Note that the Django template language doesn't use parentheses for calling methods. The `{% with %}` tag allows you to assign a value to a new variable that will be available in the template until the `{% endwith %}` tag.

The `{% with %}` template tag is useful for avoiding hitting the database or accessing expensive methods multiple times.

We use the `pluralize` template filter to display a plural suffix for the word "comment," depending on the `total_comments` value. Template filters take the value of the variable they are applied to as their input and return a computed value. We will learn more about template filters in *Chapter 3*, *Extending Your Blog Application*.

The `pluralize` template filter returns a string with the letter "s" if the value is different from `1`. The preceding text will be rendered as *0 comments*, *1 comment*, or *N comments*, depending on the number of active comments for the post.

Now, let's add the list of active comments to the post detail template.

Edit the `blog/post/detail.html` template and implement the following changes:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
  {% for comment in comments %}
    <div class="comment">
```

```
        <p class="info">
          Comment {{ forloop.counter }} by {{ comment.name }}
          {{ comment.created }}
        </p>
        {{ comment.body|linebreaks }}
      </div>
    {% empty %}
      <p>There are no comments.</p>
    {% endfor %}
{% endblock %}
```

We have added a `{% for %}` template tag to loop through the post comments. If the `comments` list is empty, we display a message that informs users that there are no comments for this post. We enumerate comments with the `{{ forloop.counter }}` variable, which contains the loop counter in each iteration. For each post, we display the name of the user who posted it, the date, and the body of the comment.

Finally, let's add the comment form to the template.

Edit the `blog/post/detail.html` template and include the comment form template as follows:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
  {% for comment in comments %}
```

```
    <div class="comment">
      <p class="info">
        Comment {{ forloop.counter }} by {{ comment.name }}
        {{ comment.created }}
      </p>
      {{ comment.body|linebreaks }}
    </div>
  {% empty %}
    <p>There are no comments.</p>
  {% endfor %}
  {% include "blog/post/includes/comment_form.html" %}
{% endblock %}
```

Open `http://127.0.0.1:8000/blog/` in your browser and click on a post title to take a look at the post detail page. You will see something like *Figure 2.22*:

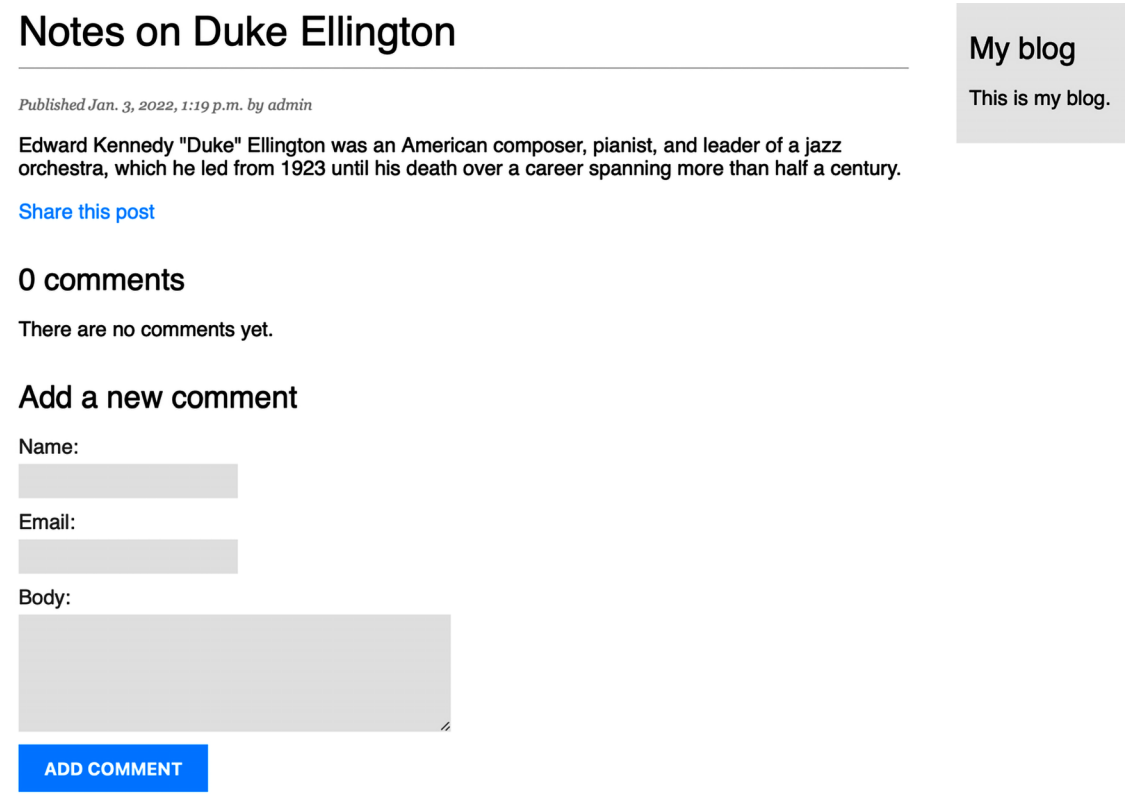


Figure 2.22: The post detail page, including the form to add a comment

Fill in the comment form with valid data and click on **Add comment**. You should see the following page:

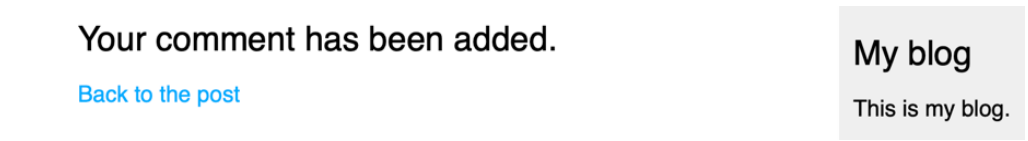


Figure 2.23: The comment added success page

Click on the **Back to the post** link. You should be redirected back to the post detail page, and you should be able to see the comment that you just added, as follows:

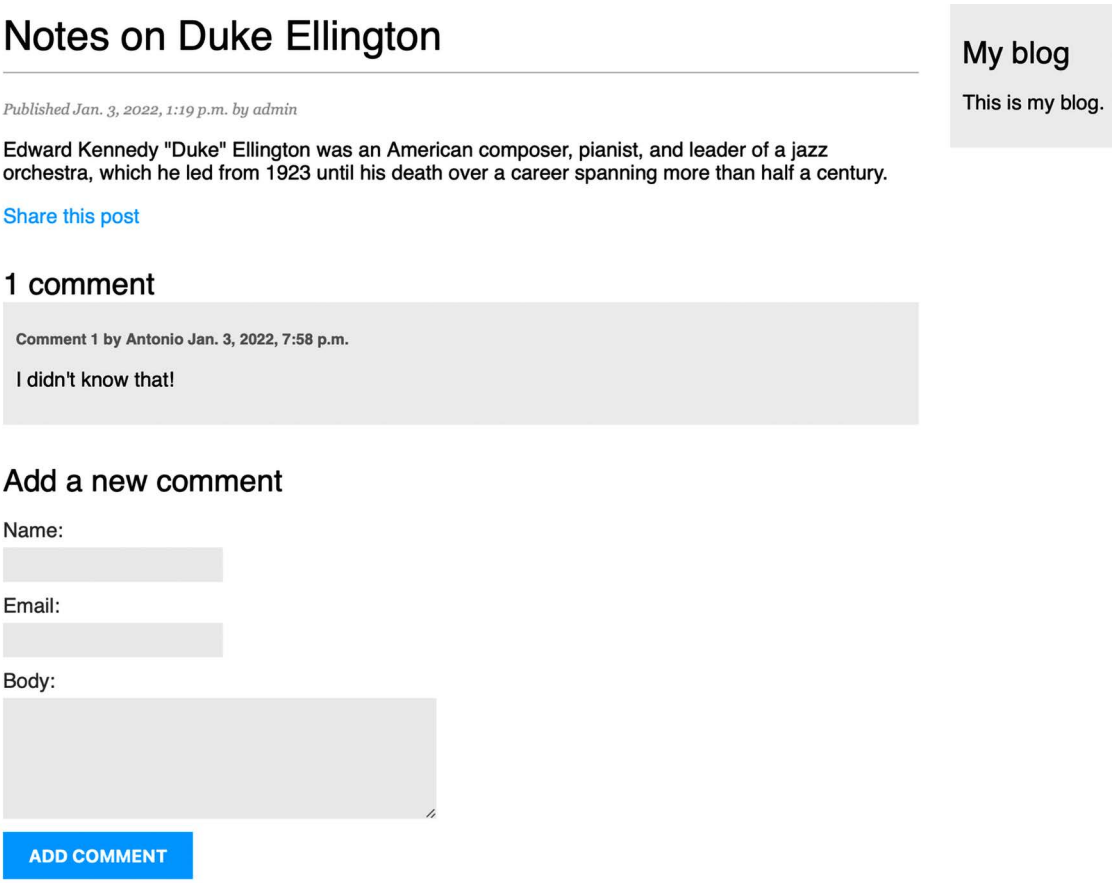


Figure 2.24: The post detail page, including a comment

Add one more comment to the post. The comments should appear below the post contents in chronological order, as follows:

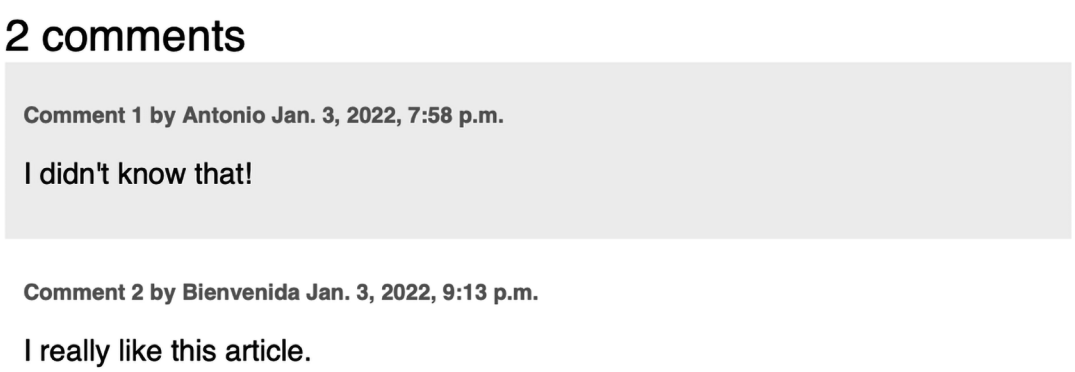


Figure 2.25: The comment list on the post detail page

Open `http://127.0.0.1:8000/admin/blog/comment/` in your browser. You will see the administration page with the list of comments you created, like this:

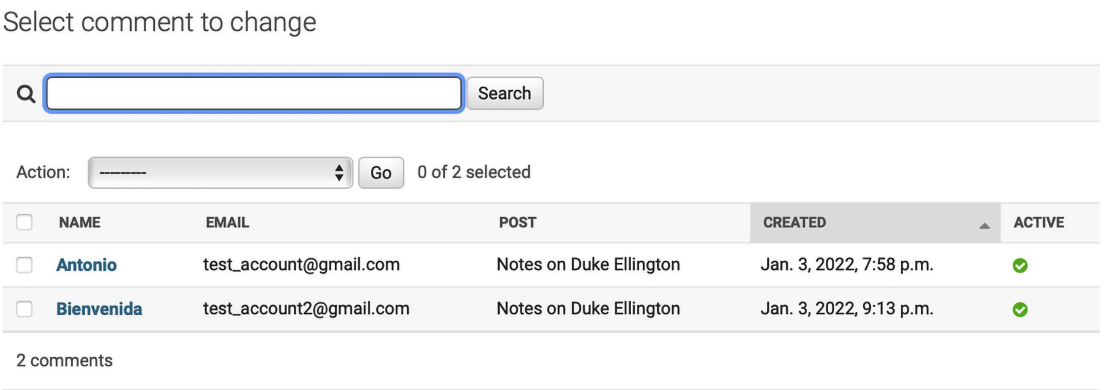


Figure 2.26: List of comments on the administration site

Click on the name of one of the posts to edit it. Uncheck the **Active** checkbox as follows and click on the **Save** button:

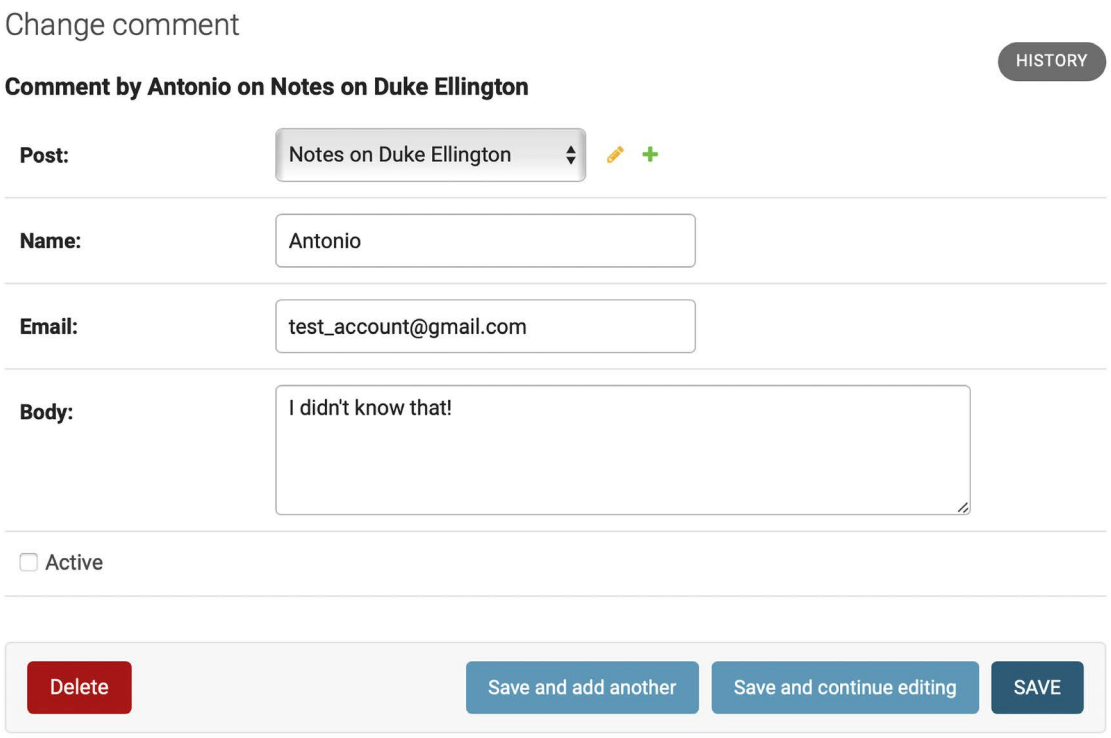


Figure 2.27: Editing a comment on the administration site

You will be redirected to the list of comments. The **Active** column will display an inactive icon for the comment, as shown in *Figure 2.28*:

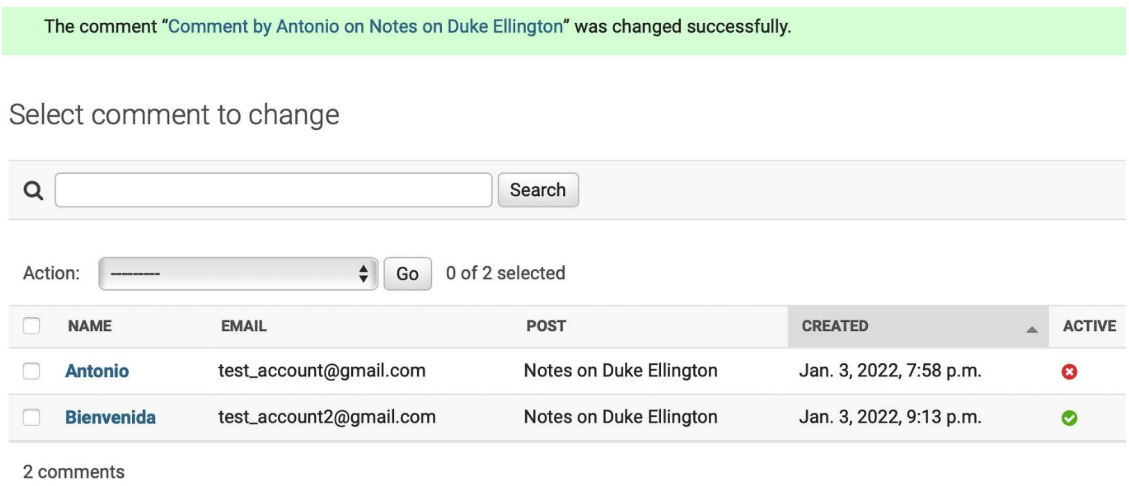


Figure 2.28: Active/inactive comments on the administration site

If you return to the post detail view, you will note that the inactive comment is no longer displayed, neither is it counted for the total number of active comments for the post:

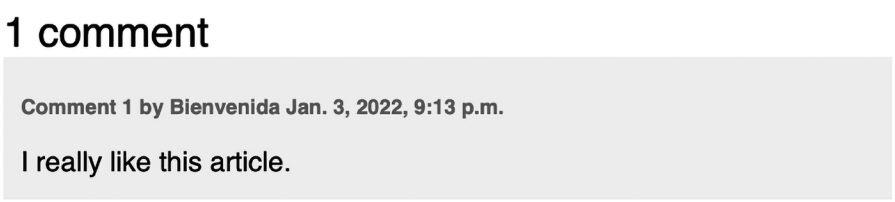


Figure 2.29: A single active comment displayed on the post detail page

Thanks to the `active` field, you can deactivate inappropriate comments and avoid showing them on your posts.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02`
- URLs utility functions – `https://docs.djangoproject.com/en/4.1/ref/urlresolvers/`
- URL path converters – `https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters`
- Django paginator class – `https://docs.djangoproject.com/en/4.1/ref/paginator/`
- Introduction to class-based views – `https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/`
- Sending emails with Django – `https://docs.djangoproject.com/en/4.1/topics/email/`
- Django form field types – `https://docs.djangoproject.com/en/4.1/ref/forms/fields/`
- Working with forms – `https://docs.djangoproject.com/en/4.1/topics/forms/`
- Creating forms from models – `https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/`
- Many-to-one model relationships – `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/`

Summary

In this chapter, you learned how to define canonical URLs for models. You created SEO-friendly URLs for blog posts, and you implemented object pagination for your post list. You also learned how to work with Django forms and model forms. You created a system to recommend posts by email and created a comment system for your blog.

In the next chapter, you will create a tagging system for the blog. You will learn how to build complex QuerySets to retrieve objects by similarity. You will learn how to create custom template tags and filters. You will also build a custom sitemap and feed for your blog posts and implement a full-text search functionality for your posts.

3

Extending Your Blog Application

The previous chapter went through the basics of forms and the creation of a comment system. You also learned how to send emails with Django. In this chapter, you will extend your blog application with other popular features used on blogging platforms, such as tagging, recommending similar posts, providing an RSS feed to readers, and allowing them to search posts. You will learn about new components and functionalities with Django by building these functionalities.

The chapter will cover the following topics:

- Integrating third-party applications
- Using `django-taggit` to implement a tagging system
- Building complex QuerySets to recommend similar posts
- Creating custom template tags and filters to show a list of the latest posts and most commented posts in the sidebar
- Creating a sitemap using the sitemap framework
- Building an RSS feed using the syndication framework
- Installing PostgreSQL
- Implementing a full-text search engine with Django and PostgreSQL

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03`.

All Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all the requirements at once with the command `pip install -r requirements.txt`.

Adding the tagging functionality

A very common functionality in blogs is to categorize posts using tags. Tags allow you to categorize content in a non-hierarchical manner, using simple keywords. A tag is simply a label or keyword that can be assigned to posts. We will create a tagging system by integrating a third-party Django tagging application into the project.

`django-taggit` is a reusable application that primarily offers you a `Tag` model and a manager to easily add tags to any model. You can take a look at its source code at `https://github.com/jazzband/django-taggit`.

First, you need to install `django-taggit` via `pip` by running the following command:

```
pip install django-taggit==3.0.0
```

Then, open the `settings.py` file of the `mysite` project and add `taggit` to your `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'taggit',
]
```

Open the `models.py` file of your `blog` application and add the `TaggableManager` manager provided by `django-taggit` to the `Post` model using the following code:

```
from taggit.managers import TaggableManager

class Post(models.Model):
    # ...
    tags = TaggableManager()
```

The `tags` manager will allow you to add, retrieve, and remove tags from `Post` objects.

The following schema shows the data models defined by `django-taggit` to create tags and store related tagged objects:

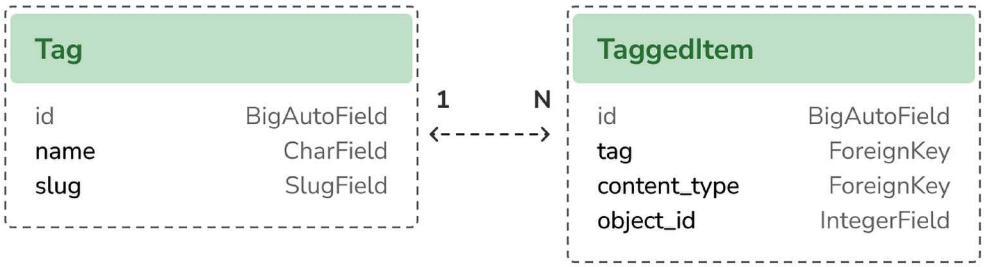


Figure 3.1: Tag models of django-taggit

The `Tag` model is used to store tags. It contains a `name` and a `slug` field.

The `TaggedItem` model is used to store the related tagged objects. It has a `ForeignKey` field for the related `Tag` object. It contains a `ForeignKey` to a `ContentType` object and an `IntegerField` to store the related `id` of the tagged object. The `content_type` and `object_id` fields combined form a generic relationship with any model in your project. This allows you to create relationships between a `Tag` instance and any other model instance of your applications. You will learn about generic relations in *Chapter 7*, *Tracking User Actions*.

Run the following command in the shell prompt to create a migration for your model changes:

```
python manage.py makemigrations blog
```

You should get the following output:

```
Migrations for 'blog':
  blog/migrations/0004_post_tags.py
    - Add field tags to post
```

Now, run the following command to create the required database tables for `django-taggit` models and to synchronize your model changes:

```
python manage.py migrate
```

You will see an output indicating that migrations have been applied, as follows:

```
Applying taggit.0001_initial... OK
Applying taggit.0002_auto_20150616_2121... OK
Applying taggit.0003_taggeditem_add_unique_index... OK
Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
Applying taggit.0005_auto_20220424_2025... OK
Applying blog.0004_post_tags... OK
```

The database is now in sync with the `taggit` models and we can start using the functionalities of `django-taggit`.

Let's now explore how to use the `tags` manager.

Open the Django shell by running the following command in the system shell prompt:

```
python manage.py shell
```

Run the following code to retrieve one of the posts (the one with the `1` ID):

```
>>> from blog.models import Post
>>> post = Post.objects.get(id=1)
```

Then, add some tags to it and retrieve its tags to check whether they were successfully added:

```
>>> post.tags.add('music', 'jazz', 'django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>, <Tag: django>]>
```

Finally, remove a tag and check the list of tags again:

```
>>> post.tags.remove('django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>]>
```

It's really easy to add, retrieve, or remove tags from a model using the manager we have defined.

Start the development server from the shell prompt with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/taggit/tag/` in your browser.

You will see the administration page with the list of `Tag` objects of the `taggit` application:

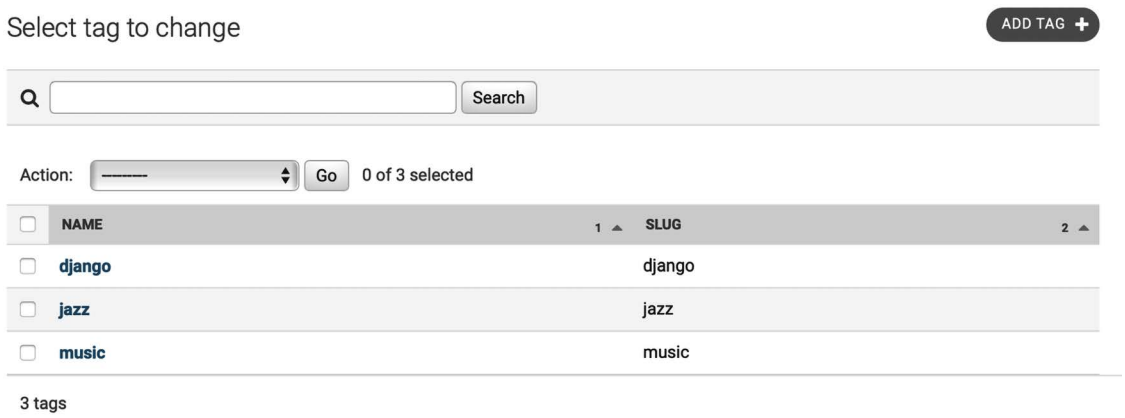


Figure 3.2: The tag change list view on the Django administration site

Click on the **jazz** tag. You will see the following:

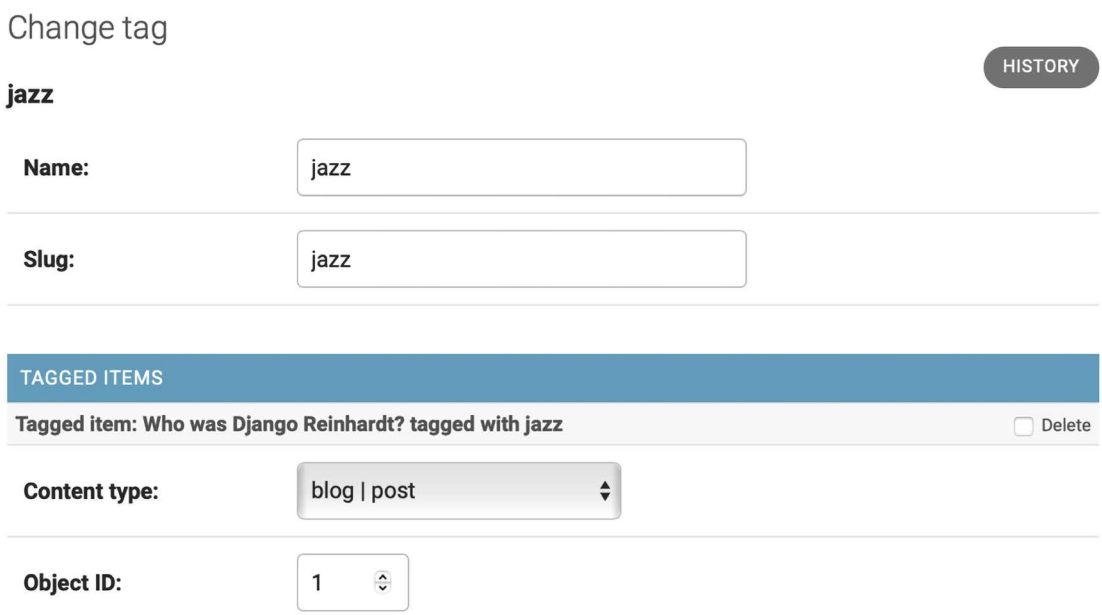


Figure 3.3: The related tags field of a Post object

Navigate to `http://127.0.0.1:8000/admin/blog/post/1/change/` to edit the post with ID `1`.

You will see that posts now include a new **Tags** field, as follows, where you can easily edit tags:

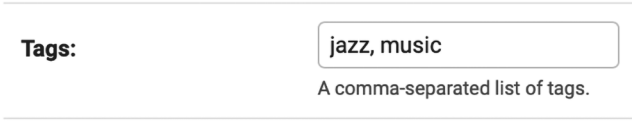


Figure 3.4: The related tags field of a Post object

Now, you need to edit your blog posts to display tags.

Open the `blog/post/list.html` template and add the following HTML code highlighted in bold:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=page_obj %}
{% endblock %}
```

The `join` template filter works the same as the Python string `join()` method to concatenate elements with the given string.

Open `http://127.0.0.1:8000/blog/` in your browser. You should be able to see the list of tags under each post title:

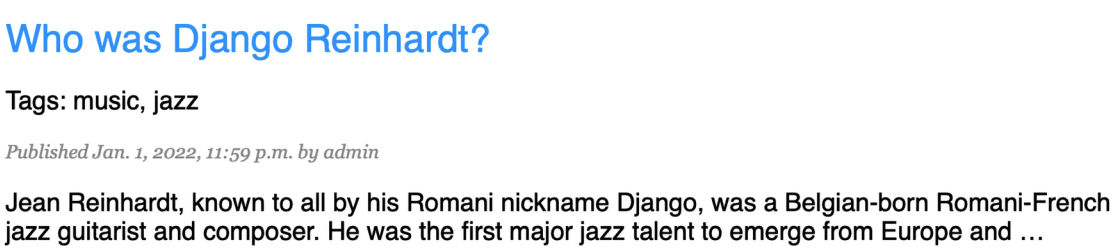


Figure 3.5: The Post list item, including related tags

Next, we will edit the `post_list` view to let users list all posts tagged with a specific tag.

Open the `views.py` file of your `blog` application, import the `Tag` model from `django-taggit`, and change the `post_list` view to optionally filter posts by a tag, as follows. New code is highlighted in bold:

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    post_list = Post.published.all()
    tag = None
    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        post_list = post_list.filter(tags__in=[tag])
    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # If page_number is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
        # If page_number is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts,
                   'tag': tag})
```

The `post_list` view now works as follows:

1. It takes an optional `tag_slug` parameter that has a `None` default value. This parameter will be passed in the URL.
2. Inside the view, we build the initial QuerySet, retrieving all published posts, and if there is a given tag slug, we get the `Tag` object with the given slug using the `get_object_or_404()` shortcut.
3. Then, we filter the list of posts by the ones that contain the given tag. Since this is a many-to-many relationship, we have to filter posts by tags contained in a given list, which, in this case, contains only one element. We use the `__in` field lookup. Many-to-many relationships occur when multiple objects of a model are associated with multiple objects of another model. In our application, a post can have multiple tags and a tag can be related to multiple posts. You will learn how to create many-to-many relationships in *Chapter 6*, *Sharing Content on Your Website*. You can discover more about many-to-many relationships at `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/`.
4. Finally, the `render()` function now passes the new `tag` variable to the template.

Remember that QuerySets are lazy. The QuerySets to retrieve posts will only be evaluated when you loop over the post list when rendering the template.

Open the `urls.py` file of your `blog` application, comment out the class-based `PostListView` URL pattern, and uncomment the `post_list` view, like this:

```
path('', views.post_list, name='post_list'),
# path('', views.PostListView.as_view(), name='post_list'),
```

Add the following additional URL pattern to list posts by tag:

```
path('tag/<slug:tag_slug>/',
     views.post_list, name='post_list_by_tag'),
```

As you can see, both patterns point to the same view, but they have different names. The first pattern will call the `post_list` view without any optional parameters, whereas the second pattern will call the view with the `tag_slug` parameter. You use a `slug` path converter to match the parameter as a lowercase string with ASCII letters or numbers, plus the hyphen and underscore characters.

The `urls.py` file of the `blog` application should now look like this:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
         views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
]
```

Since you are using the `post_list` view, edit the `blog/post/list.html` template and modify the pagination to use the `posts` object:

```
{% include "pagination.html" with page=posts %}
```

Add the following lines highlighted in bold to the `blog/post/list.html` template:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
  {% endif %}
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

If a user is accessing the blog, they will see the list of all posts. If they filter by posts tagged with a specific tag, they will see the tag that they are filtering by.

Now, edit the `blog/post/list.html` template and change the way tags are displayed, as follows. New lines are highlighted in bold:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
  {% endif %}
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
```

```
        </a>
      </h2>
      <p class="tags">
        Tags:
        {% for tag in post.tags.all %}
          <a href="{% url "blog:post_list_by_tag" tag.slug %}">
            {{ tag.name }}
          </a>
          {% if not forloop.last %}, {% endif %}
        {% endfor %}
      </p>
      <p class="date">
        Published {{ post.publish }} by {{ post.author }}
      </p>
      {{ post.body|truncatewords:30|linebreaks }}
    {% endfor %}
    {% include "pagination.html" with page=posts %}
  {% endblock %}
```

In the preceding code, we loop through all the tags of a post displaying a custom link to the URL to filter posts by that tag. We build the URL with `{% url "blog:post_list_by_tag" tag.slug %}`, using the name of the URL and the `slug` tag as its parameter. You separate the tags by commas.

Open `http://127.0.0.1:8000/blog/tag/jazz/` in your browser. You will see the list of posts filtered by that tag, like this:

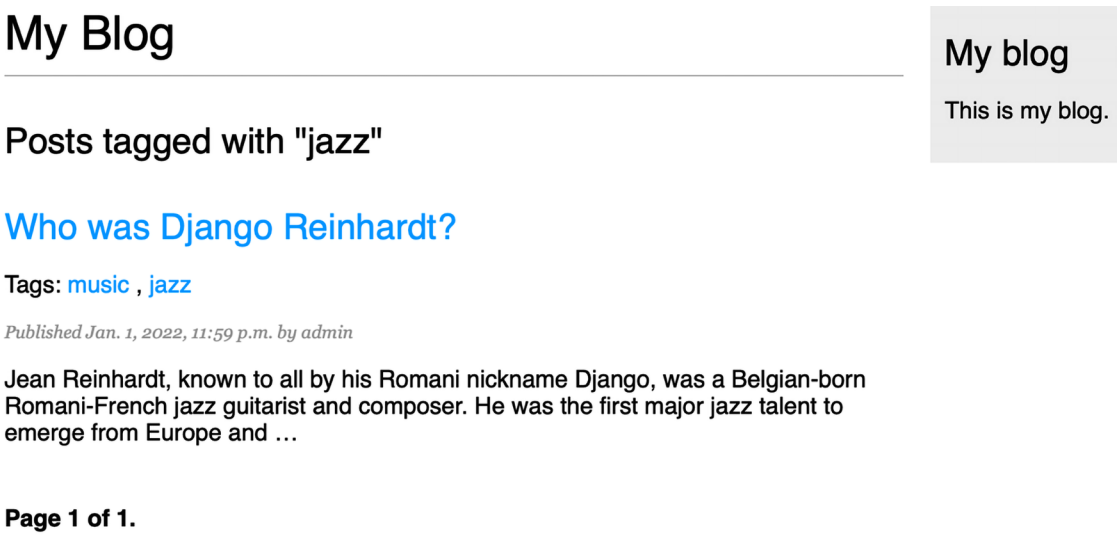


Figure 3.6: A post filtered by the tag "jazz"

Retrieving posts by similarity

Now that we have implemented tagging for blog posts, you can do many interesting things with tags. Tags allow you to categorize posts in a non-hierarchical manner. Posts about similar topics will have several tags in common. We will build a functionality to display similar posts by the number of tags they share. In this way, when a user reads a post, we can suggest to them that they read other related posts.

In order to retrieve similar posts for a specific post, you need to perform the following steps:

1. Retrieve all tags for the current post
2. Get all posts that are tagged with any of those tags
3. Exclude the current post from that list to avoid recommending the same post
4. Order the results by the number of tags shared with the current post
5. In the case of two or more posts with the same number of tags, recommend the most recent post
6. Limit the query to the number of posts you want to recommend

These steps are translated into a complex QuerySet that you will include in your `post_detail` view.

Open the `views.py` file of your `blog` application and add the following import at the top of it:

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

This is the `Count` aggregation function of the Django ORM. This function will allow you to perform aggregated counts of tags. `django.db.models` includes the following aggregation functions:

- `Avg`: The mean value
- `Max`: The maximum value
- `Min`: The minimum value
- `Count`: The total number of objects

You can learn about aggregation at `https://docs.djangoproject.com/en/4.1/topics/db/aggregation/`.

Open the `views.py` file of your `blog` application and add the following lines to the `post_detail` view. New lines are highlighted in bold:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)

    # List of active comments for this post
    comments = post.comments.filter(active=True)
```

```
    # Form for users to comment
    form = CommentForm()

    # List of similar posts
    post_tags_ids = post.tags.values_list('id', flat=True)
    similar_posts = Post.published.filter(tags__in=post_tags_ids)\
                                  .exclude(id=post.id)
    similar_posts = similar_posts.annotate(same_tags=Count('tags'))\
                                 .order_by('-same_tags','-publish')[:4]

    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                   'comments': comments,
                   'form': form,
                   'similar_posts': similar_posts})
```

The preceding code is as follows:

1. You retrieve a Python list of IDs for the tags of the current post. The `values_list()` QuerySet returns tuples with the values for the given fields. You pass `flat=True` to it to get single values such as `[1, 2, 3, ...]` instead of one-tuples such as `[(1,), (2,), (3,) ...]`.
2. You get all posts that contain any of these tags, excluding the current post itself.
3. You use the `Count` aggregation function to generate a calculated field—`same_tags`—that contains the number of tags shared with all the tags queried.
4. You order the result by the number of shared tags (descending order) and by `publish` to display recent posts first for the posts with the same number of shared tags. You slice the result to retrieve only the first four posts.
5. We pass the `similar_posts` object to the context dictionary for the `render()` function.

Now, edit the `blog/post/detail.html` template and add the following code highlighted in bold:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
```

```
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>

  <h2>Similar posts</h2>
  {% for post in similar_posts %}
    <p>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </p>
  {% empty %}
    There are no similar posts yet.
  {% endfor %}

  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
  {% for comment in comments %}
    <div class="comment">
      <p class="info">
        Comment {{ forloop.counter }} by {{ comment.name }}
        {{ comment.created }}
      </p>
      {{ comment.body|linebreaks }}
    </div>
  {% empty %}
    <p>There are no comments yet.</p>
  {% endfor %}
  {% include "blog/post/includes/comment_form.html" %}
{% endblock %}
```

The post detail page should look like this:

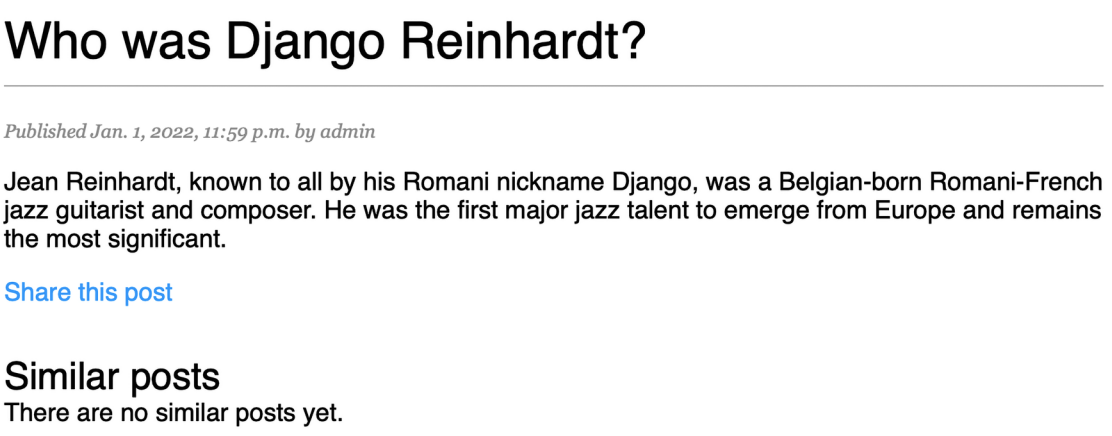


Figure 3.7: The post detail page, including a list of similar posts

Open `http://127.0.0.1:8000/admin/blog/post/` in your browser, edit a post that has no tags, and add the `music` and `jazz` tags as follows:

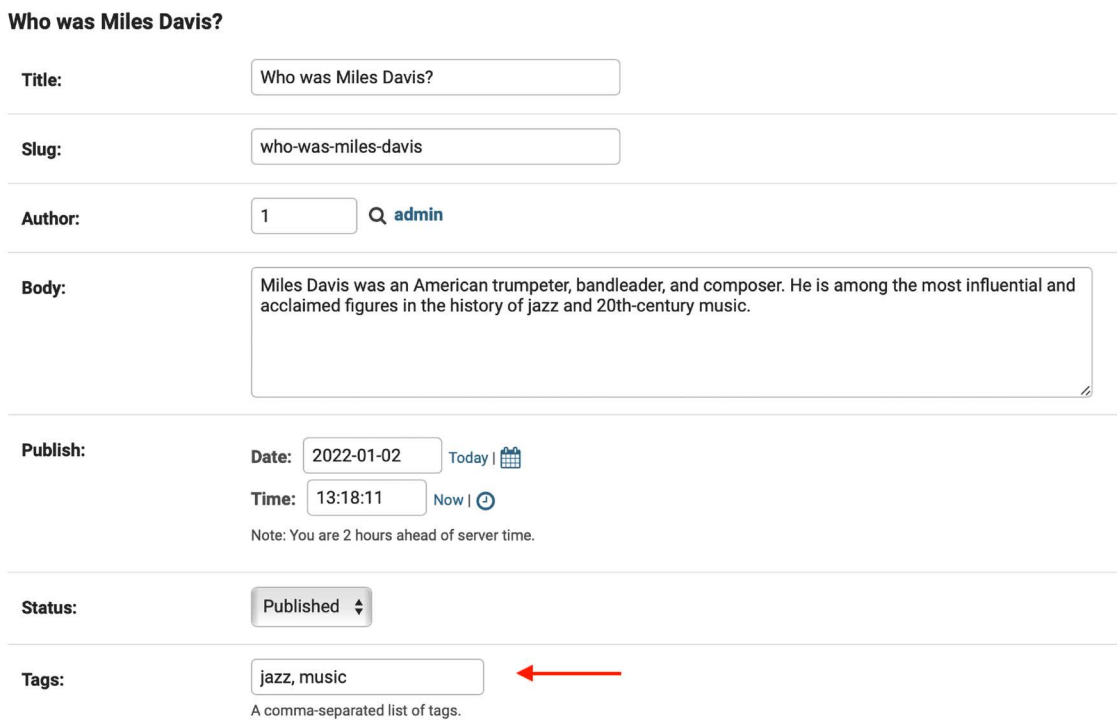


Figure 3.8: Adding the "jazz" and "music" tags to a post

Edit another post and add the `jazz` tag as follows:

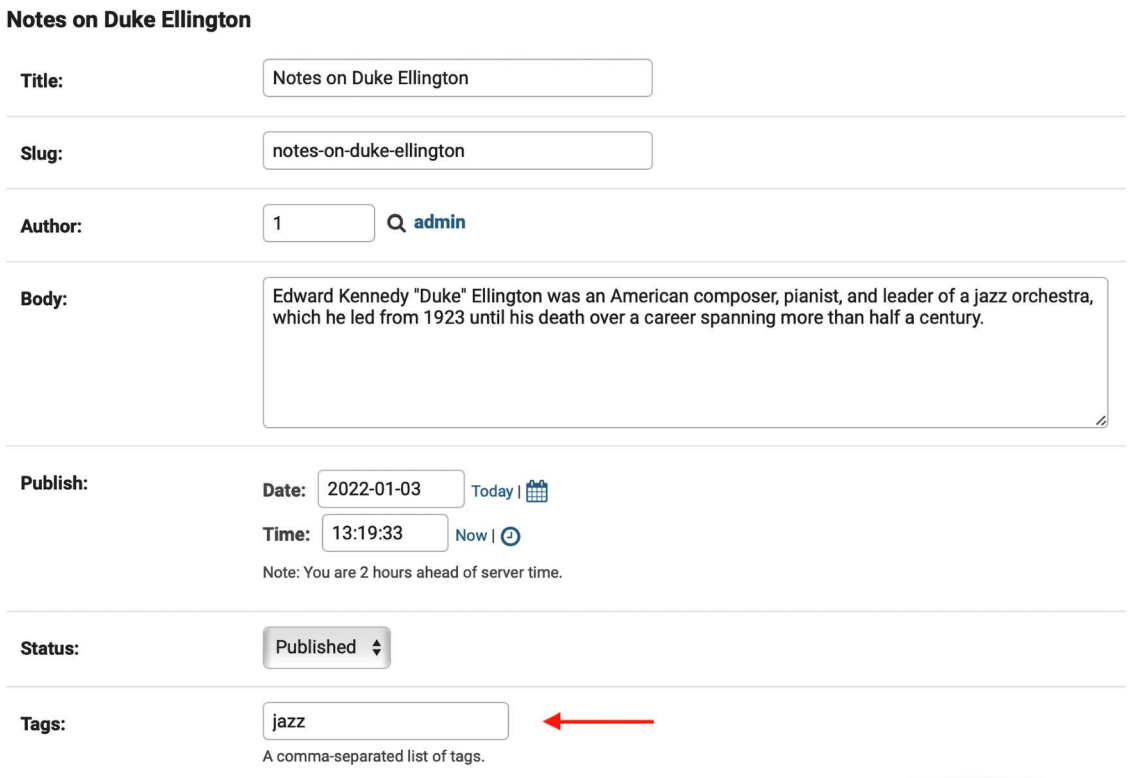


Figure 3.9: Adding the "jazz" tag to a post

The post detail page for the first post should now look like this:

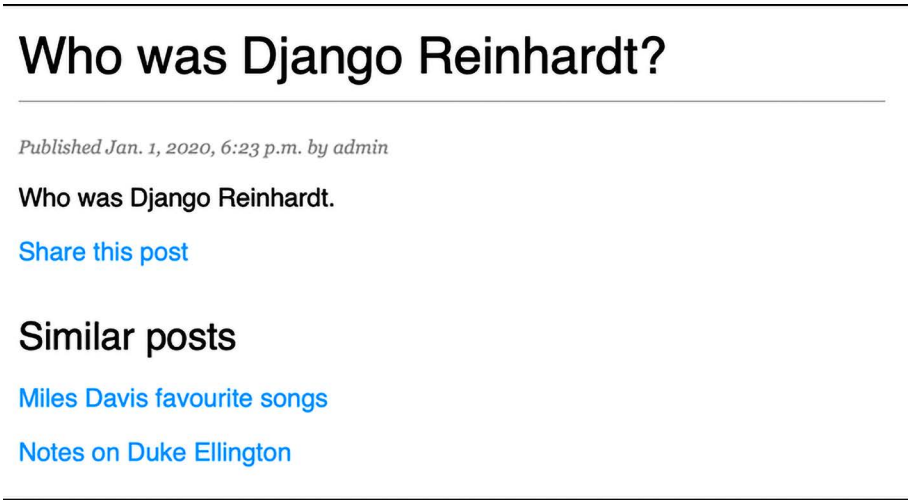


Figure 3.10: The post detail page, including a list of similar posts

The posts recommended in the **Similar posts** section of the page appear in descending order based on the number of shared tags with the original post.

We are now able to successfully recommend similar posts to the readers. `django-taggit` also includes a `similar_objects()` manager that you can use to retrieve objects by shared tags. You can take a look at all `django-taggit` managers at `https://django-taggit.readthedocs.io/en/latest/api.html`.

You can also add the list of tags to your post detail template in the same way as you did in the `blog/post/list.html` template.

Creating custom template tags and filters

Django offers a variety of built-in template tags, such as `{% if %}` or `{% block %}`. You used different template tags in *Chapter 1*, *Building a Blog Application*, and *Chapter 2*, *Enhancing Your Blog with Advanced Features*. You can find a complete reference of built-in template tags and filters at `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/`.

Django also allows you to create your own template tags to perform custom actions. Custom template tags come in very handy when you need to add a functionality to your templates that is not covered by the core set of Django template tags. This can be a tag to execute a QuerySet or any server-side processing that you want to reuse across templates. For example, we could build a template tag to display the list of latest posts published on the blog. We could include this list in the sidebar, so that it is always visible, regardless of the view that processes the request.

Implementing custom template tags

Django provides the following helper functions that allow you to easily create template tags:

- `simple_tag`: Processes the given data and returns a string
- `inclusion_tag`: Processes the given data and returns a rendered template

Template tags must live inside Django applications.

Inside your `blog` application directory, create a new directory, name it `templatetags`, and add an empty `__init__.py` file to it. Create another file in the same folder and name it `blog_tags.py`. The file structure of the blog application should look like the following:

```
blog/
    __init__.py
    models.py
    ...
    templatetags/
        __init__.py
        blog_tags.py
```

The way you name the file is important. You will use the name of this module to load tags in templates.

Creating a simple template tag

Let's start by creating a simple tag to retrieve the total posts that have been published on the blog.

Edit the `templatetags/blog_tags.py` file you just created and add the following code:

```
from django import template
from ..models import Post

register = template.Library()

@register.simple_tag
def total_posts():
    return Post.published.count()
```

We have created a simple template tag that returns the number of posts published in the blog.

Each module that contains template tags needs to define a variable called `register` to be a valid tag library. This variable is an instance of `template.Library`, and it's used to register the template tags and filters of the application.

In the preceding code, we have defined a tag called `total_posts` with a simple Python function. We have added the `@register.simple_tag` decorator to the function, to register it as a simple tag. Django will use the function's name as the tag name. If you want to register it using a different name, you can do so by specifying a `name` attribute, such as `@register.simple_tag(name='my_tag')`.

After adding a new template tags module, you will need to restart the Django development server in order to use the new tags and filters in templates.

Before using custom template tags, we have to make them available for the template using the `{% load %}` tag. As mentioned before, we need to use the name of the Python module containing your template tags and filters.

Edit the `blog/templates/base.html` template and add `{% load blog_tags %}` at the top of it to load your template tags module. Then, use the tag you created to display your total posts, as follows. The new lines are highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
```

```
    <link href="{% static "css/blog.css" %}" rel="stylesheet">
  </head>
  <body>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
    <div id="sidebar">
      <h2>My blog</h2>
      <p>
        This is my blog.
        I've written {% total_posts %} posts so far.
      </p>
    </div>
  </body>
  </html>
```

You will need to restart the server to keep track of the new files added to the project. Stop the development server with *Ctrl* + *C* and run it again using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/blog/` in your browser. You should see the total number of posts in the sidebar of the site, as follows:

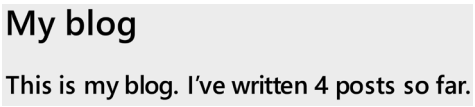


Figure 3.11: The total posts published included in the sidebar

If you see the following error message, it's very likely you didn't restart the development server:

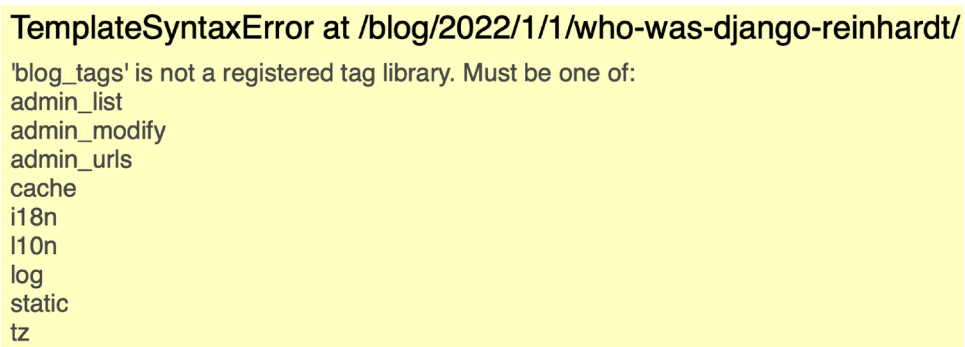


Figure 3.12: The error message when a template tag library is not registered

Template tags allow you to process any data and add it to any template regardless of the view executed. You can perform QuerySets or process any data to display results in your templates.

Creating an inclusion template tag

We will create another tag to display the latest posts in the sidebar of the blog. This time, we will implement an inclusion tag. Using an inclusion tag, you can render a template with context variables returned by your template tag.

Edit the `templatetags/blog_tags.py` file and add the following code:

```
@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[:count]
    return {'latest_posts': latest_posts}
```

In the preceding code, we have registered the template tag using the `@register.inclusion_tag` decorator. We have specified the template that will be rendered with the returned values using `blog/post/latest_posts.html`. The template tag will accept an optional `count` parameter that defaults to `5`. This parameter will allow us to specify the number of posts to display. We use this variable to limit the results of the query `Post.published.order_by('-publish')[:count]`.

Note that the function returns a dictionary of variables instead of a simple value. Inclusion tags have to return a dictionary of values, which is used as the context to render the specified template. The template tag we just created allows us to specify the optional number of posts to display as `{% show_latest_posts 3 %}`.

Now, create a new template file under `blog/post/` and name it `latest_posts.html`.

Edit the new `blog/post/latest_posts.html` template and add the following code to it:

```
<ul>
  {% for post in latest_posts %}
    <li>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </li>
  {% endfor %}
</ul>
```

In the preceding code, you display an unordered list of posts using the `latest_posts` variable returned by your template tag. Now, edit the `blog/base.html` template and add the new template tag to display the last three posts, as follows. The new lines are highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
```

```
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>
      This is my blog.
      I've written {% total_posts %} posts so far.
    </p>
    <h3>Latest posts</h3>
    {% show_latest_posts 3 %}
  </div>
</body>
</html>
```

The template tag is called, passing the number of posts to display, and the template is rendered in place with the given context.

Next, return to your browser and refresh the page. The sidebar should now look like this:

Figure 3.13: The blog sidebar, including the latest published posts

Creating a template tag that returns a QuerySet

Finally, we will create a simple template tag that returns a value. We will store the result in a variable that can be reused, rather than outputting it directly. We will create a tag to display the most commented posts.

Edit the `templatetags/blog_tags.py` file and add the following import and template tag to it:

```
from django.db.models import Count

@register.simple_tag
def get_most_commented_posts(count=5):
    return Post.published.annotate(
               total_comments=Count('comments')
           ).order_by('-total_comments')[:count]
```

In the preceding template tag, you build a QuerySet using the `annotate()` function to aggregate the total number of comments for each post. You use the `Count` aggregation function to store the number of comments in the computed `total_comments` field for each `Post` object. You order the QuerySet by the computed field in descending order. You also provide an optional `count` variable to limit the total number of objects returned.

In addition to `Count`, Django offers the aggregation functions `Avg`, `Max`, `Min`, and `Sum`. You can read more about aggregation functions at `https://docs.djangoproject.com/en/4.1/topics/db/aggregation/`.

Next, edit the `blog/base.html` template and add the following code highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>
      This is my blog.
      I've written {% total_posts %} posts so far.
    </p>
    <h3>Latest posts</h3>
    {% show_latest_posts 3 %}
    <h3>Most commented posts</h3>
```

```
    {% get_most_commented_posts as most_commented_posts %}
    <ul>
      {% for post in most_commented_posts %}
        <li>
          <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
</body>
</html>
```

In the preceding code, we store the result in a custom variable using the `as` argument followed by the variable name. For the template tag, we use `{% get_most_commented_posts as most_commented_posts %}` to store the result of the template tag in a new variable named `most_commented_posts`. Then, we display the returned posts using an HTML unordered list element.

Now open your browser and refresh the page to see the final result. It should look like the following:

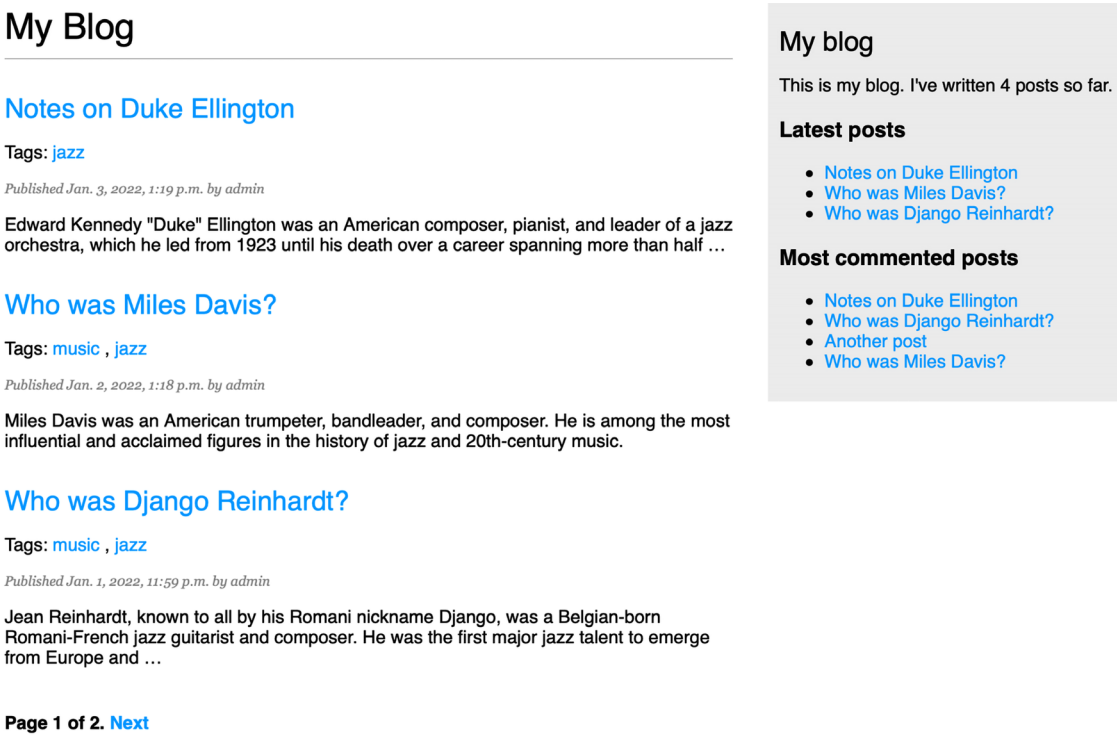


Figure 3.14: The post list view, including the complete sidebar with the latest and most commented posts

You have now a clear idea about how to build custom template tags. You can read more about them at https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/.

Implementing custom template filters

Django has a variety of built-in template filters that allow you to alter variables in templates. These are Python functions that take one or two parameters, the value of the variable that the filter is applied to, and an optional argument. They return a value that can be displayed or treated by another filter.

A filter is written like `{{ variable|my_filter }}`. Filters with an argument are written like `{{ variable|my_filter:"foo" }}`. For example, you can use the `capfirst` filter to capitalize the first character of the value, like `{{ value|capfirst }}`. If `value` is `django`, the output will be `Django`. You can apply as many filters as you like to a variable, for example, `{{ variable|filter1|filter2 }}`, and each filter will be applied to the output generated by the preceding filter.

You can find the list of Django's built-in template filters at https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#built-in-filter-reference.

Creating a template filter to support Markdown syntax

We will create a custom filter to enable you to use Markdown syntax in your blog posts and then convert the post body to HTML in the templates.

Markdown is a plain text formatting syntax that is very simple to use, and it's intended to be converted into HTML. You can write posts using simple Markdown syntax and get the content automatically converted into HTML code. Learning Markdown syntax is much easier than learning HTML. By using Markdown, you can get other non-tech savvy contributors to easily write posts for your blog. You can learn the basics of the Markdown format at https://daringfireball.net/projects/markdown/basics.

First, install the Python `markdown` module via `pip` using the following command in the shell prompt:

```
pip install markdown==3.4.1
```

Then, edit the `templatetags/blog_tags.py` file and include the following code:

```
from django.utils.safestring import mark_safe
import markdown

@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

We register template filters in the same way as template tags. To prevent a name clash between the function name and the `markdown` module, we have named the function `markdown_format` and we have named the filter `markdown` for use in templates, such as `{{ variable|markdown }}`.

Django escapes the HTML code generated by filters; characters of HTML entities are replaced with their HTML encoded characters. For example, `<p>` is converted to `&lt;p&gt;` (*less than* symbol, *p* character, *greater than* symbol).

We use the `mark_safe` function provided by Django to mark the result as safe HTML to be rendered in the template. By default, Django will not trust any HTML code and will escape it before placing it in the output. The only exceptions are variables that are marked as safe from escaping. This behavior prevents Django from outputting potentially dangerous HTML and allows you to create exceptions for returning safe HTML.

Edit the `blog/post/detail.html` template and add the following new code highlighted in bold:

```
{% extends "blog/base.html" %}
{% load blog_tags %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|markdown }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>

  <h2>Similar posts</h2>
  {% for post in similar_posts %}
    <p>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </p>
  {% empty %}
    There are no similar posts yet.
  {% endfor %}

  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
  {% for comment in comments %}
    <div class="comment">
      <p class="info">
```

```
        Comment {{ forloop.counter }} by {{ comment.name }}
        {{ comment.created }}
      </p>
      {{ comment.body|linebreaks }}
    </div>
  {% empty %}
    <p>There are no comments yet.</p>
  {% endfor %}

  {% include "blog/post/includes/comment_form.html" %}
{% endblock %}
```

We have replaced the `linebreaks` filter of the `{{ post.body }}` template variable with the `markdown` filter. This filter will not only transform line breaks into `<p>` tags; it will also transform Markdown formatting into HTML.

Edit the `blog/post/list.html` template and add the following new code highlighted in bold:

```
{% extends "blog/base.html" %}
{% load blog_tags %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
  {% endif %}
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="tags">
      Tags:
      {% for tag in post.tags.all %}
        <a href="{% url "blog:post_list_by_tag" tag.slug %}">
          {{ tag.name }}
        </a>
        {% if not forloop.last %}, {% endif %}
      {% endfor %}
```

```
    </p>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|markdown|truncatewords_html:30 }}
  {% endfor %}
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

We have added the new `markdown` filter to the `{{ post.body }}` template variable. This filter will transform the Markdown content into HTML. Therefore, we have replaced the previous `truncatewords` filter with the `truncatewords_html` filter. This filter truncates a string after a certain number of words avoiding unclosed HTML tags.

Now open `http://127.0.0.1:8000/admin/blog/post/add/` in your browser and create a new post with the following body:

```
This is a post formatted with markdown
---------------------------------------

*This is emphasized* and **this is more emphasized**.

Here is a list:

* One
* Two
* Three

And a [link to the Django website](https://www.djangoproject.com/).
```

The form should look like this:

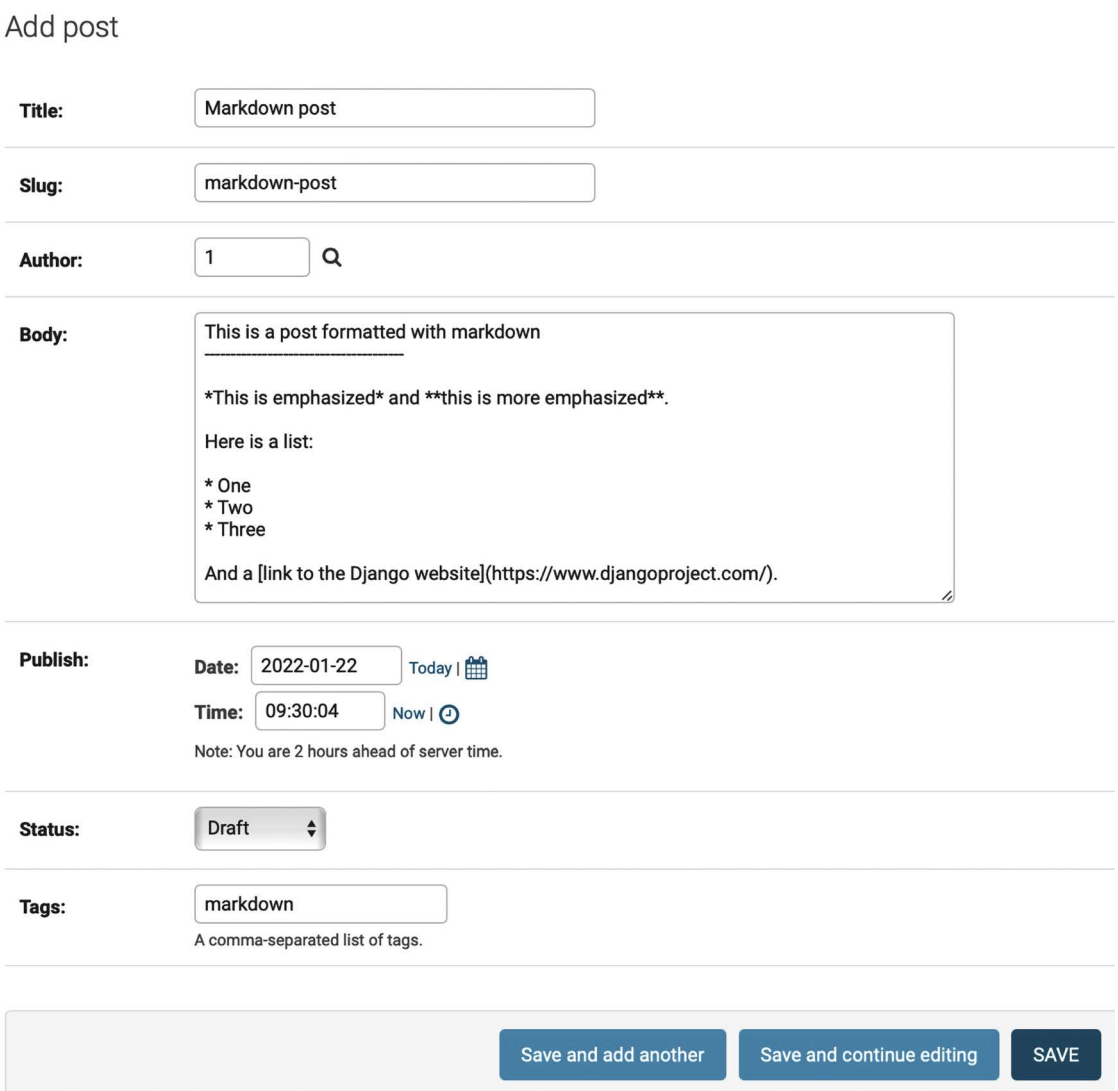


Figure 3.15: The post with Markdown content rendered as HTML

Open `http://127.0.0.1:8000/blog/` in your browser and take a look at how the new post is rendered. You should see the following output:

Figure 3.16: The post with Markdown content rendered as HTML

As you can see in *Figure 3.16*, custom template filters are very useful for customizing formatting. You can find more information about custom filters at `https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/#writing-custom-template-filters`.

Adding a sitemap to the site

Django comes with a sitemap framework, which allows you to generate sitemaps for your site dynamically. A sitemap is an XML file that tells search engines the pages of your website, their relevance, and how frequently they are updated. Using a sitemap will make your site more visible in search engine rankings because it helps crawlers to index your website's content.

The Django sitemap framework depends on `django.contrib.sites`, which allows you to associate objects to particular websites that are running with your project. This comes in handy when you want to run multiple sites using a single Django project. To install the sitemap framework, we will need to activate both the `sites` and the `sitemap` applications in your project.

Edit the `settings.py` file of the project and add `django.contrib.sites` and `django.contrib.sitemaps` to the `INSTALLED_APPS` setting. Also, define a new setting for the site ID, as follows. New code is highlighted in bold:

```
# ...

SITE_ID = 1

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'taggit',
    'django.contrib.sites',
    'django.contrib.sitemaps',
]
```

Now, run the following command from the shell prompt to create the tables of the Django site application in the database:

```
python manage.py migrate
```

You should see an output that contains the following lines:

```
Applying sites.0001_initial... OK
Applying sites.0002_alter_domain_unique... OK
```

The `sites` application is now synced with the database.

Next, create a new file inside your `blog` application directory and name it `sitemaps.py`. Open the file and add the following code to it:

```
from django.contrib.sitemaps import Sitemap
from .models import Post

class PostSitemap(Sitemap):
    changefreq = 'weekly'
    priority = 0.9
```

```
    def items(self):
        return Post.published.all()

    def lastmod(self, obj):
        return obj.updated
```

We have defined a custom sitemap by inheriting the `Sitemap` class of the `sitemaps` module. The `changefreq` and `priority` attributes indicate the change frequency of your post pages and their relevance in your website (the maximum value is `1`).

The `items()` method returns the QuerySet of objects to include in this sitemap. By default, Django calls the `get_absolute_url()` method on each object to retrieve its URL. Remember that we implemented this method in *Chapter 2*, *Enhancing Your Blog with Advanced Features*, to define the canonical URL for posts. If you want to specify the URL for each object, you can add a `location` method to your sitemap class.

The `lastmod` method receives each object returned by `items()` and returns the last time the object was modified.

Both the `changefreq` and `priority` attributes can be either methods or attributes. You can take a look at the complete sitemap reference in the official Django documentation located at `https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/`.

We have created the sitemap. Now we just need to create an URL for it.

Edit the main `urls.py` file of the `mysite` project and add the sitemap, as follows. New lines are highlighted in bold:

```
from django.urls import path, include
from django.contrib import admin
from django.contrib.sitemaps.views import sitemap
from blog.sitemaps import PostSitemap

sitemaps = {
    'posts': PostSitemap,
}

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
    path('sitemap.xml', sitemap, {'sitemaps': sitemaps},
         name='django.contrib.sitemaps.views.sitemap')
]
```

In the preceding code, we have included the required imports and have defined a `sitemaps` dictionary. Multiple sitemaps can be defined for the site. We have defined a URL pattern that matches with the `sitemap.xml` pattern and uses the `sitemap` view provided by Django. The `sitemaps` dictionary is passed to the `sitemap` view.

Start the development from the shell prompt with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/sitemap.xml` in your browser. You will see an XML output including all of the published posts like this:

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>http://example.com/blog/2022/1/22/markdown-post/</loc>
    <lastmod>2022-01-22</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/3/notes-on-duke-ellington/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreqa>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/2/who-was-miles-davis/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/who-was-django-reinhardt/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/another-post/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

The URL for each `Post` object is built by calling its `get_absolute_url()` method.

The `lastmod` attribute corresponds to the post `updated` date field, as you specified in your sitemap, and the `changefreq` and `priority` attributes are also taken from the `PostSitemap` class.

The domain used to build the URLs is `example.com`. This domain comes from a `Site` object stored in the database. This default object was created when you synced the site's framework with your database. You can read more about the `sites` framework at `https://docs.djangoproject.com/en/4.1/ref/contrib/sites/`.

Open `http://127.0.0.1:8000/admin/sites/site/` in your browser. You should see something like this:

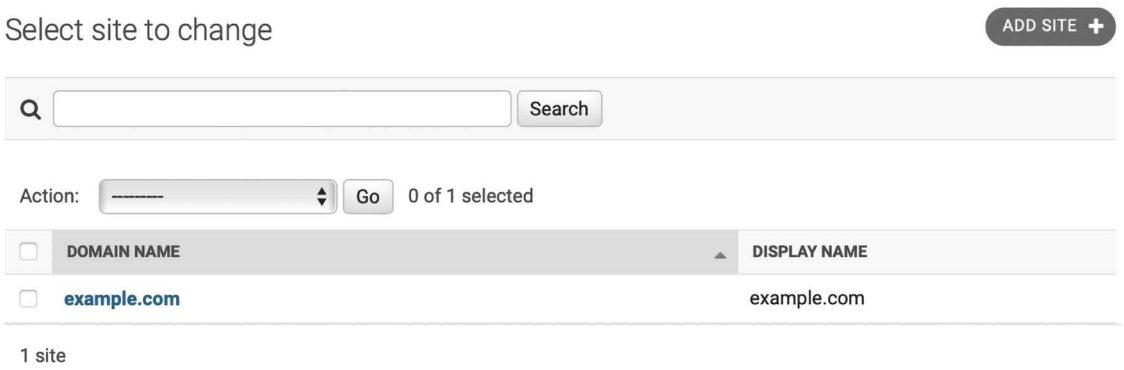


Figure 3.17: The Django administration list view for the Site model of the site's framework

Figure 3.17 contains the list display administration view for the site's framework. Here, you can set the domain or host to be used by the site's framework and the applications that depend on it. To generate URLs that exist in your local environment, change the domain name to `localhost:8000`, as shown in *Figure 3.18*, and save it:

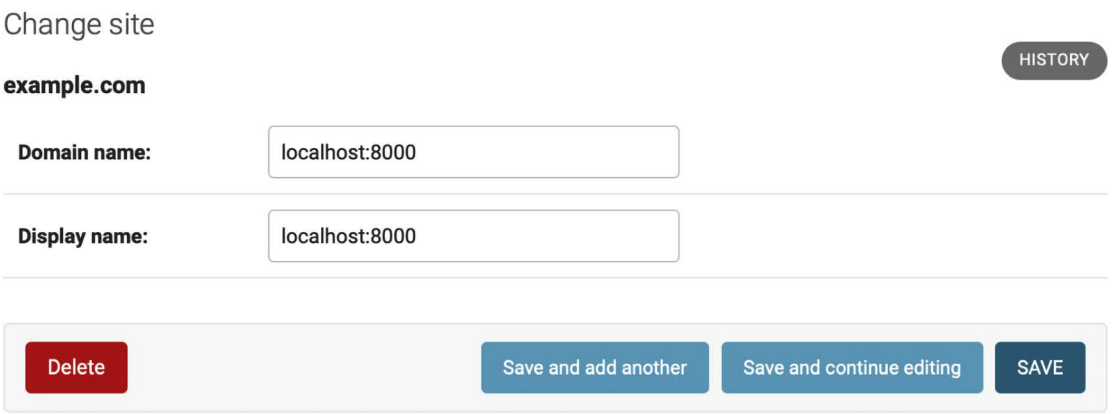


Figure 3.18: The Django administration edit view for the Site model of the site's framework

Open `http://127.0.0.1:8000/sitemap.xml` in your browser again. The URLs displayed in your feed will now use the new hostname and look like `http://localhost:8000/blog/2022/1/22/markdown-post/`. Links are now accessible in your local environment. In a production environment, you will have to use your website's domain to generate absolute URLs.

Creating feeds for blog posts

Django has a built-in syndication feed framework that you can use to dynamically generate RSS or Atom feeds in a similar manner to creating sitemaps using the site's framework. A web feed is a data format (usually XML) that provides users with the most recently updated content. Users can subscribe to the feed using a feed aggregator, a software that is used to read feeds and get new content notifications.

Create a new file in your `blog` application directory and name it `feeds.py`. Add the following lines to it:

```
import markdown
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords_html
from django.urls import reverse_lazy
from .models import Post

class LatestPostsFeed(Feed):
    title = 'My blog'
    link = reverse_lazy('blog:post_list')
    description = 'New posts of my blog.'

    def items(self):
        return Post.published.all()[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return truncatewords_html(markdown.markdown(item.body), 30)

    def item_pubdate(self, item):
        return item.publish
```

In the preceding code, we have defined a feed by subclassing the `Feed` class of the syndication framework. The `title`, `link`, and `description` attributes correspond to the `<title>`, `<link>`, and `<description>` RSS elements, respectively.

We use `reverse_lazy()` to generate the URL for the `link` attribute. The `reverse()` method allows you to build URLs by their name and pass optional parameters. We used `reverse()` in *Chapter 2*, *Enhancing Your Blog with Advanced Features*.

The `reverse_lazy()` utility function is a lazily evaluated version of `reverse()`. It allows you to use a URL reversal before the project's URL configuration is loaded.

The `items()` method retrieves the objects to be included in the feed. We retrieve the last five published posts to include them in the feed.

The `item_title()`, `item_description()`, and `item_pubdate()` methods will receive each object returned by `items()` and return the title, description and publication date for each item.

In the `item_description()` method, we use the `markdown()` function to convert Markdown content to HTML and the `truncatewords_html()` template filter function to cut the description of posts after 30 words, avoiding unclosed HTML tags.

Now, edit the `blog/urls.py` file, import the `LatestPostsFeed` class, and instantiate the feed in a new URL pattern, as follows. New lines are highlighted in bold:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
         views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
    path('feed/', LatestPostsFeed(), name='post_feed'),
]
```

Navigate to `http://127.0.0.1:8000/blog/feed/` in your browser. You should now see the RSS feed, including the last five blog posts:

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
  <channel>
    <title>My blog</title>
```

```
    <link>http://localhost:8000/blog/</link>
    <description>New posts of my blog.</description>
    <atom:link href="http://localhost:8000/blog/feed/" rel="self"/>
    <language>en-us</language>
    <lastBuildDate>Fri, 2 Jan 2020 09:56:40 +0000</lastBuildDate>
    <item>
      <title>Who was Django Reinhardt?</title>
      <link>http://localhost:8000/blog/2020/1/2/who-was-django-
      reinhardt/</link>
      <description>Who was Django Reinhardt.</description>
      <guid>http://localhost:8000/blog/2020/1/2/who-was-django-
      reinhardt/</guid>
    </item>
    ...
  </channel>
</rss>
```

If you use Chrome, you will see the XML code. If you use Safari, it will ask you to install an RSS feed reader.

Let's install an RSS desktop client to view the RSS feed with a user-friendly interface. We will use Fluent Reader, which is a multi-platform RSS reader.

Download Fluent Reader for Linux, macOS, or Windows from `https://github.com/yang991178/fluent-reader/releases`.

Install Fluent Reader and open it. You will see the following screen:

Figure 3.19: Fluent Reader with no RSS feed sources

Click on the settings icon on the top right of the window. You will see a screen to add RSS feed sources like the following one:

Figure 3.20: Adding an RSS feed in Fluent Reader

Enter `http://127.0.0.1:8000/blog/feed/` in the **Add source** field and click on the **Add** button.

You will see a new entry with the RSS feed of the blog in the table below the form, like this:

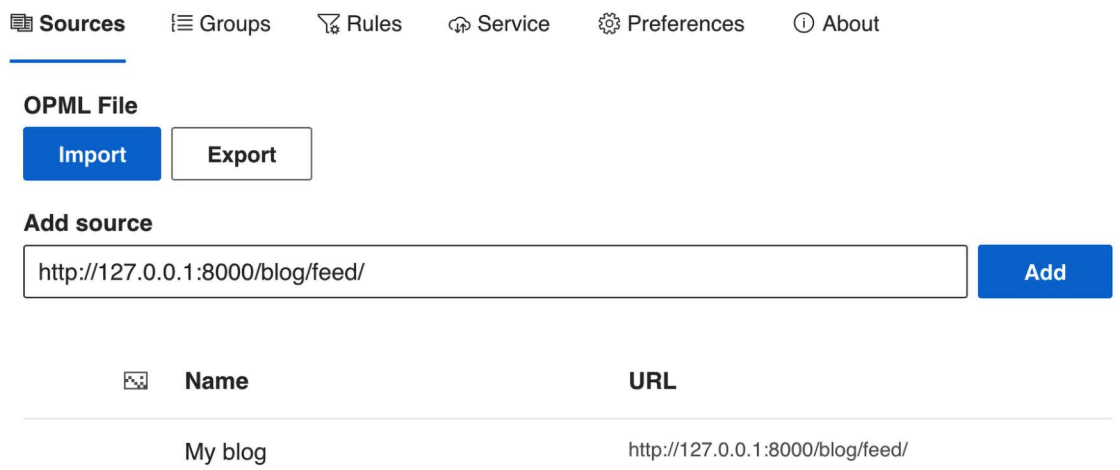


Figure 3.21: RSS feed sources in Fluent Reader

Now, go back to the main screen of Fluent Reader. You should be able to see the posts included in the blog RSS feed, as follows:

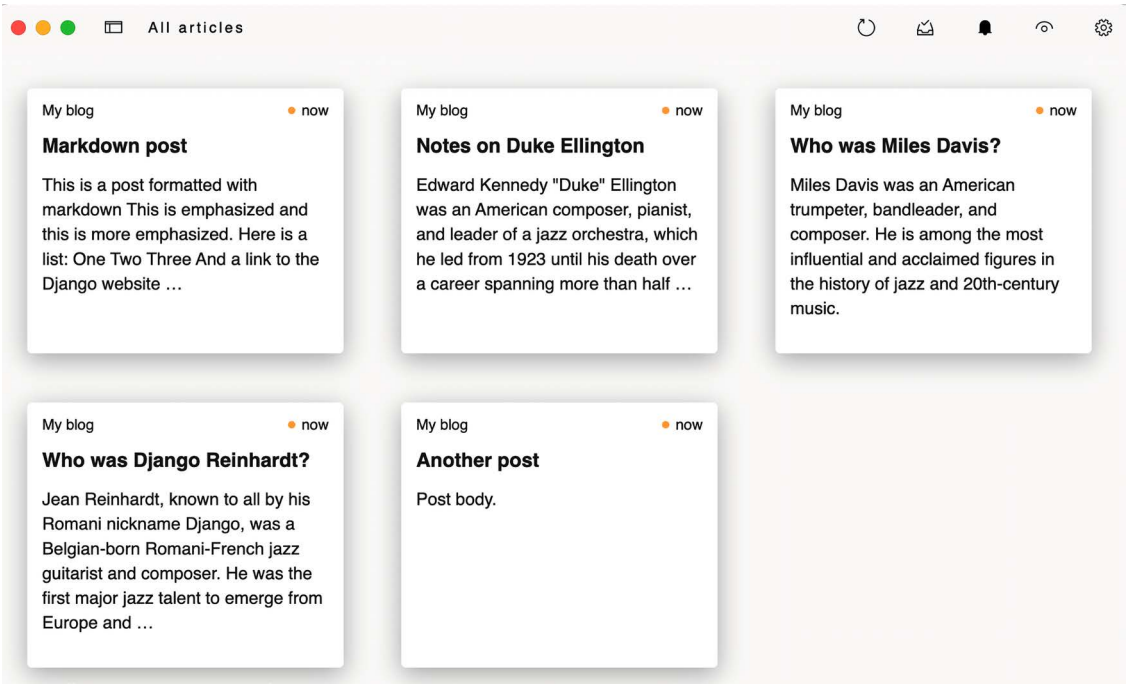


Figure 3.22: RSS feed of the blog in Fluent Reader

Click on a post to see a description:

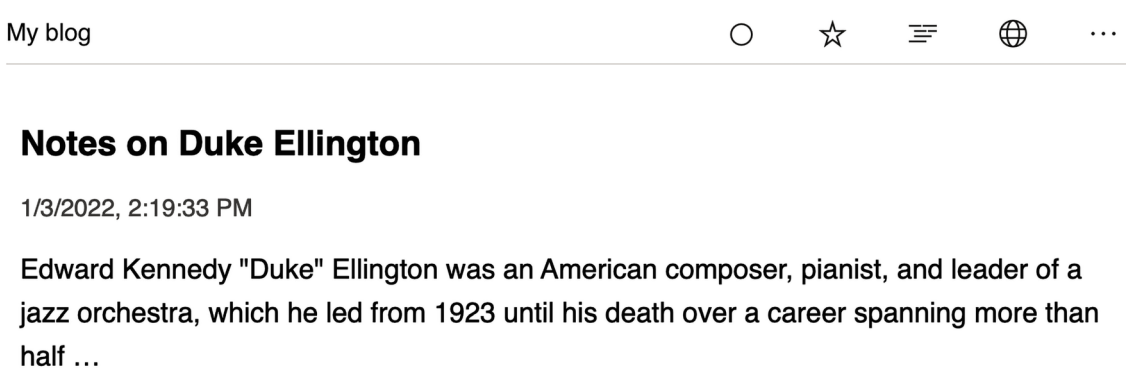


Figure 3.23: The post description in Fluent Reader

Click on the third icon at the top right of the window to load the full content of the post page:

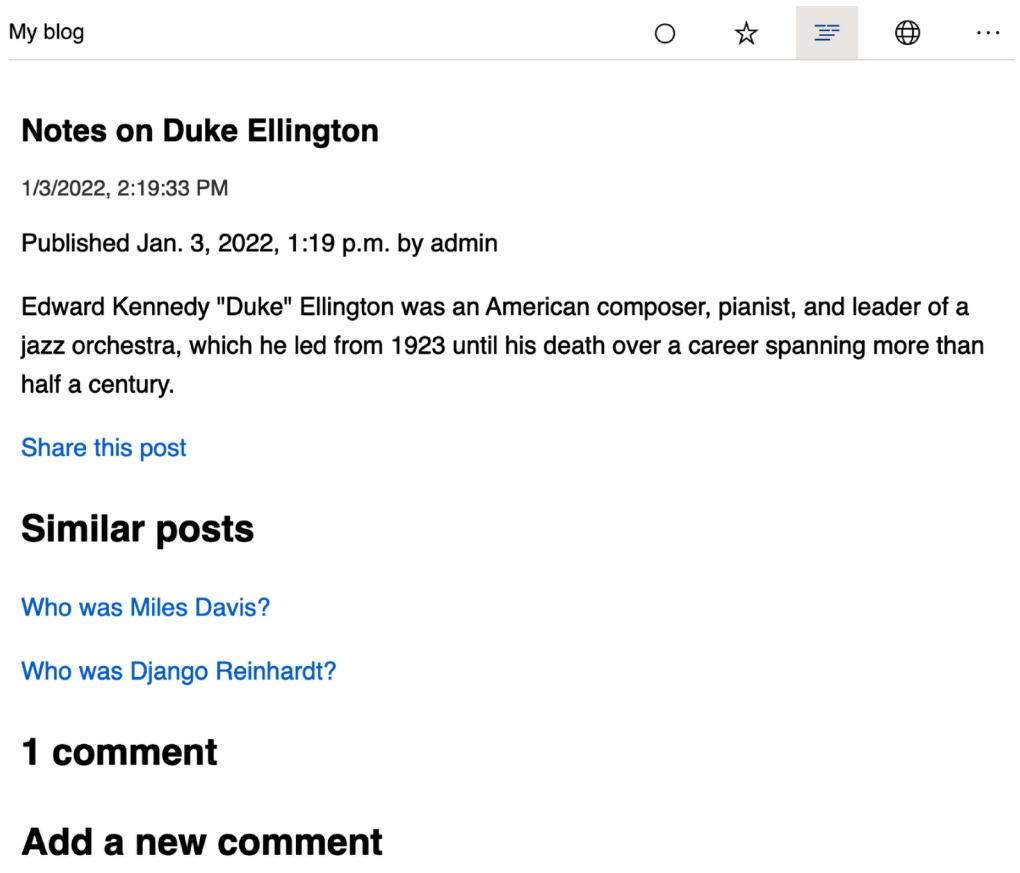


Figure 3.24: The full content of a post in Fluent Reader

The final step is to add an RSS feed subscription link to the blog's sidebar.

Open the `blog/base.html` template and add the following code highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
```

```
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>
      This is my blog.
      I've written {% total_posts %} posts so far.
    </p>
    <p>
      <a href="{% url "blog:post_feed" %}">
        Subscribe to my RSS feed
      </a>
    </p>
    <h3>Latest posts</h3>
    {% show_latest_posts 3 %}
    <h3>Most commented posts</h3>
    {% get_most_commented_posts as most_commented_posts %}
    <ul>
      {% for post in most_commented_posts %}
        <li>
          <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
</body>
</html>
```

Now open `http://127.0.0.1:8000/blog/` in your browser and take a look at the sidebar. The new link will take users to the blog's feed:

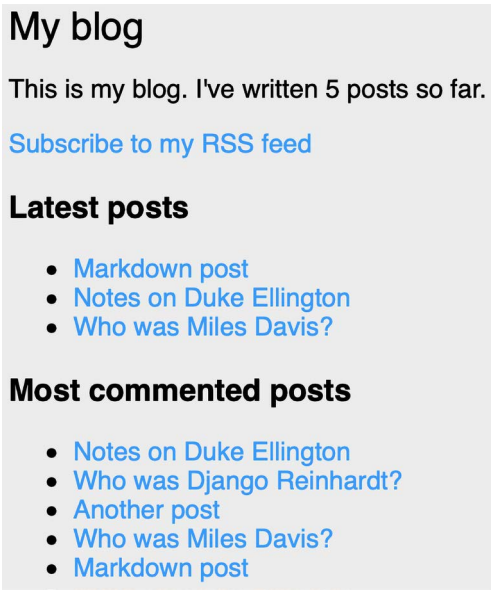


Figure 3.25: The RSS feed subscription link added to the sidebar

You can read more about the Django syndication feed framework at `https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/`.

Adding full-text search to the blog

Next, we will add search capabilities to the blog. Searching for data in the database with user input is a common task for web applications. The Django ORM allows you to perform simple matching operations using, for example, the `contains` filter (or its case-insensitive version, `icontains`). You can use the following query to find posts that contain the word `framework` in their body:

```
from blog.models import Post
Post.objects.filter(body__contains='framework')
```

However, if you want to perform complex search lookups, retrieving results by similarity, or by weighting terms based on how frequently they appear in the text or by how important different fields are (for example, relevancy of the term appearing in the title versus in the body), you will need to use a full-text search engine. When you consider large blocks of text, building queries with operations on a string of characters is not enough. A full-text search examines the actual words against stored content as it tries to match search criteria.

Django provides a powerful search functionality built on top of PostgreSQL's full-text search features. The `django.contrib.postgres` module provides functionalities offered by PostgreSQL that are not shared by the other databases that Django supports. You can learn about PostgreSQL's full-text search support at `https://www.postgresql.org/docs/14/textsearch.html`.

Although Django is a database-agnostic web framework, it provides a module that supports part of the rich feature set offered by PostgreSQL, which is not offered by other databases that Django supports.

Installing PostgreSQL

We are currently using an SQLite database for the `mysite` project. SQLite support for full-text search is limited and Django doesn't support it out of the box. However, PostgreSQL is much better suited for full-text search and we can use the `django.contrib.postgres` module to use PostgreSQL's full-text search capabilities. We will migrate our data from SQLite to PostgreSQL to benefit from its full-text search features.

SQLite is sufficient for development purposes. However, for a production environment, you will need a more powerful database, such as PostgreSQL, MariaDB, MySQL, or Oracle.

Download the PostgreSQL installer for macOS or Windows at `https://www.postgresql.org/download/`. On the same page, you can find instructions to install PostgreSQL on different Linux distributions. Follow the instructions on the website to install and run PostgreSQL.

If you are using macOS and you choose to install PostgreSQL using `Postgres.app`, you will need to configure the `$PATH` variable to use the command line tools, as explained in `https://postgresapp.com/documentation/cli-tools.html`.

You also need to install the `psycopg2` PostgreSQL adapter for Python. Run the following command in the shell prompt to install it:

```
pip install psycopg2-binary==2.9.3
```

Creating a PostgreSQL database

Let's create a user for the PostgreSQL database. We will use `psql`, which is a terminal-based frontend to PostgreSQL. Enter the PostgreSQL terminal by running the following command in the shell prompt:

```
psql
```

You will see the following output:

```
psql (14.2)
Type "help" for help.
```

Enter the following command to create a user that can create databases:

```
CREATE USER blog WITH PASSWORD 'xxxxxx';
```

Replace xxxxxx with your desired password and execute the command. You will see the following output:

```
CREATE ROLE
```

The user has been created. Let's now create a `blog` database and give ownership to the `blog` user you just created.

Execute the following command:

```
CREATE DATABASE blog OWNER blog ENCODING 'UTF8';
```

With this command we tell PostgreSQL to create a database named `blog`, we give the ownership of the database to the `blog` user we created before, and we indicate that the `UTF8` encoding has to be used for the new database. You will see the following output:

```
CREATE DATABASE
```

We have successfully created the PostgreSQL user and database.

Dumping the existing data

Before switching the database in the Django project, we need to dump the existing data from the SQLite database. We will export the data, switch the project's database to PostgreSQL, and import the data into the new database.

Django comes with a simple way to load and dump data from the database into files that are called **fixtures**. Django supports fixtures in JSON, XML, or YAML formats. We are going to create a fixture with all data contained in the database.

The `dumpdata` command dumps data from the database into the standard output, serialized in JSON format by default. The resulting data structure includes information about the model and its fields for Django to be able to load it into the database.

You can limit the output to the models of an application by providing the application names to the command, or specifying single models for outputting data using the `app.Model` format. You can also specify the format using the `--format` flag. By default, `dumpdata` outputs the serialized data to the standard output. However, you can indicate an output file using the `--output` flag. The `--indent` flag allows you to specify indentation. For more information on `dumpdata` parameters, run `python manage.py dumpdata --help`.

Execute the following command from the shell prompt:

```
python manage.py dumpdata --indent=2 --output=mysite_data.json
```

You will see an output similar to the following:

```
[..............................................]
```

All existing data has been exported in JSON format to a new file named `mysite_data.json`. You can view the file contents to see the JSON structure that includes all the different data objects for the different models of your installed applications. If you get an encoding error when running the command, include the `-Xutf8` flag as follows to activate Python UTF-8 mode:

```
python -Xutf8 manage.py dumpdata --indent=2 --output=mysite_data.json
```

We will now switch the database in the Django project and then we will import the data into the new database.

Switching the database in the project

Edit the `settings.py` file of your project and modify the `DATABASES` setting to make it look as follows. New code is highlighted in bold:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'blog',
        'USER': 'blog',
        'PASSWORD': 'xxxxxx',
    }
}
```

Replace xxxxxx with the password you used when creating the PostgreSQL user. The new database is empty.

Run the following command to apply all database migrations to the new PostgreSQL database:

```
python manage.py migrate
```

You will see an output, including all the migrations that have been applied, like this:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions, sites,
taggit
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
```

```
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying taggit.0001_initial... OK
  Applying taggit.0002_auto_20150616_2121... OK
  Applying taggit.0003_taggeditem_add_unique_index... OK
  Applying blog.0001_initial... OK
  Applying blog.0002_alter_post_slug... OK
  Applying blog.0003_comment... OK
  Applying blog.0004_post_tags... OK
  Applying sessions.0001_initial... OK
  Applying sites.0001_initial... OK
  Applying sites.0002_alter_domain_unique... OK
  Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
  Applying taggit.0005_auto_20220424_2025... OK
```

Loading the data into the new database

Run the following command to load the data into the PostgreSQL database:

```
python manage.py loaddata mysite_data.json
```

You will see the following output:

```
Installed 104 object(s) from 1 fixture(s)
```

The number of objects might differ, depending on the users, posts, comments, and other objects that have been created in the database.

Start the development server from the shell prompt with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/blog/post/` in your browser to verify that all posts have been loaded into the new database. You should see all the posts, as follows:

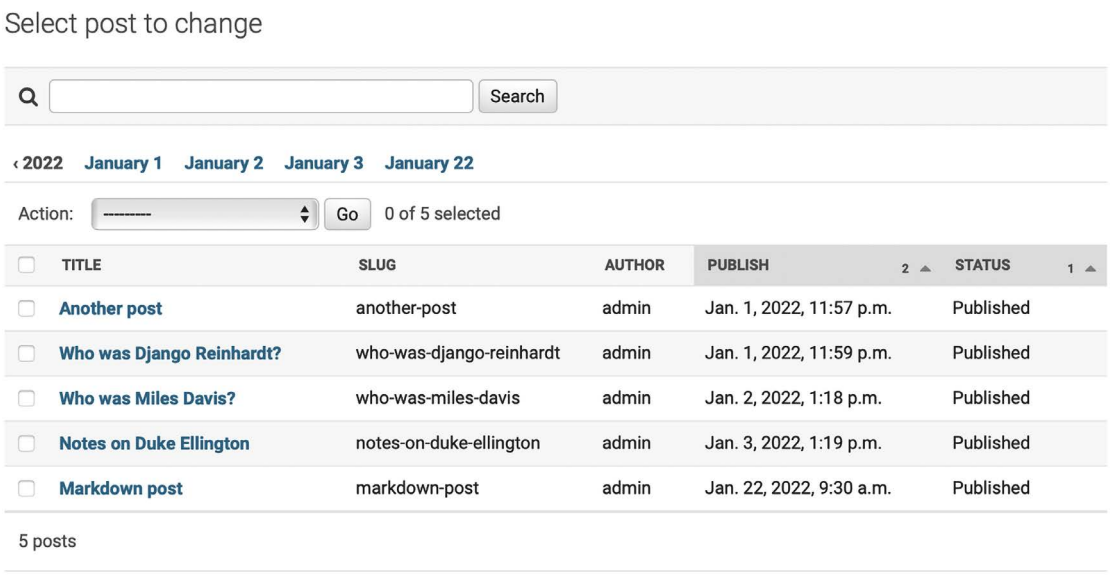


Figure 3.26: The list of posts on the administration site

Simple search lookups

Edit the `settings.py` file of your project and add `django.contrib.postgres` to the `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'taggit',
    'django.contrib.sites',
    'django.contrib.sitemaps',
    'django.contrib.postgres',
]
```

Open the Django shell by running the following command in the system shell prompt:

```
python manage.py shell
```

Now you can search against a single field using the `search` QuerySet lookup.

Run the following code in the Python shell:

```
>>> from blog.models import Post
>>> Post.objects.filter(title__search='django')
<QuerySet [<Post: Who was Django Reinhardt?>]>
```

This query uses PostgreSQL to create a search vector for the `body` field and a search query from the term `django`. Results are obtained by matching the query with the vector.

Searching against multiple fields

You might want to search against multiple fields. In this case, you will need to define a `SearchVector` object. Let's build a vector that allows you to search against the `title` and `body` fields of the `Post` model.

Run the following code in the Python shell:

```
>>> from django.contrib.postgres.search import SearchVector
>>> from blog.models import Post
>>>
>>> Post.objects.annotate(
...     search=SearchVector('title', 'body'),
... ).filter(search='django')
<QuerySet [<Post: Markdown post>, <Post: Who was Django Reinhardt?>]>
```

Using `annotate` and defining `SearchVector` with both fields, you provide a functionality to match the query against both the `title` and `body` of the posts.

Full-text search is an intensive process. If you are searching for more than a few hundred rows, you should define a functional index that matches the search vector you are using. Django provides a `SearchVectorField` field for your models. You can read more about this at `https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/#performance`.

Building a search view

Now, you will create a custom view to allow your users to search posts. First, you will need a search form. Edit the `forms.py` file of the `blog` application and add the following form:

```
class SearchForm(forms.Form):
    query = forms.CharField()
```

You will use the `query` field to let users introduce search terms. Edit the `views.py` file of the `blog` application and add the following code to it:

```
# ...
from django.contrib.postgres.search import SearchVector
from .forms import EmailPostForm, CommentForm, SearchForm

# ...

def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            results = Post.published.annotate(
                search=SearchVector('title', 'body'),
            ).filter(search=query)

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

In the preceding view, first, we instantiate the `SearchForm` form. To check whether the form is submitted, we look for the `query` parameter in the `request.GET` dictionary. We send the form using the `GET` method instead of `POST` so that the resulting URL includes the `query` parameter and is easy to share. When the form is submitted, we instantiate it with the submitted `GET` data, and verify that the form data is valid. If the form is valid, we search for published posts with a custom `SearchVector` instance built with the `title` and `body` fields.

The search view is now ready. We need to create a template to display the form and the results when the user performs a search.

Create a new file inside the `templates/blog/post/` directory, name it `search.html`, and add the following code to it:

```
{% extends "blog/base.html" %}
{% load blog_tags %}
```

```
{% block title %}Search{% endblock %}

{% block content %}
  {% if query %}
    <h1>Posts containing "{{ query }}"</h1>
    <h3>
      {% with results.count as total_results %}
        Found {{ total_results }} result{{ total_results|pluralize }}
      {% endwith %}
    </h3>
    {% for post in results %}
      <h4>
        <a href="{{ post.get_absolute_url }}">
          {{ post.title }}
        </a>
      </h4>
      {{ post.body|markdown|truncatewords_html:12 }}
    {% empty %}
      <p>There are no results for your query.</p>
    {% endfor %}
    <p><a href="{% url "blog:post_search" %}">Search again</a></p>
  {% else %}
    <h1>Search for posts</h1>
    <form method="get">
      {{ form.as_p }}
      <input type="submit" value="Search">
    </form>
  {% endif %}
{% endblock %}
```

As in the search view, we distinguish whether the form has been submitted by the presence of the `query` parameter. Before the query is submitted, we display the form and a submit button. When the search form is submitted, we display the query performed, the total number of results, and the list of posts that match the search query.

Finally, edit the `urls.py` file of the `blog` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
```

```
    path('tag/<slug:tag_slug>/',
         views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
    path('feed/', LatestPostsFeed(), name='post_feed'),
    path('search/', views.post_search, name='post_search'),
]
```

Next, open `http://127.0.0.1:8000/blog/search/` in your browser. You should see the following search form:

Figure 3.27: The form with the query field to search for posts

Enter a query and click on the **SEARCH** button. You will see the results of the search query, as follows:

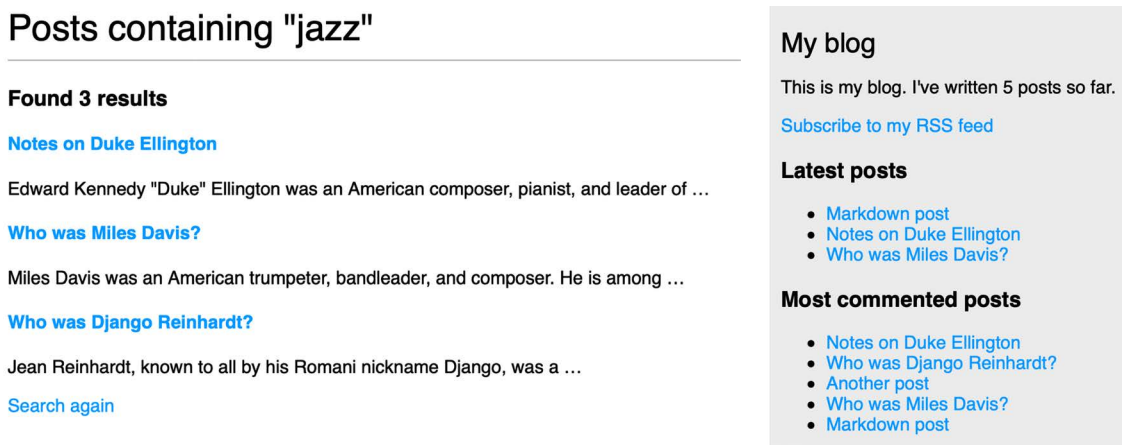


Figure 3.28: Search results for the term "jazz"

Congratulations! You have created a basic search engine for your blog.

Stemming and ranking results

Stemming is the process of reducing words to their word stem, base, or root form. Stemming is used by search engines to reduce indexed words to their stem, and to be able to match inflected or derived words. For example, the words "music", "musical" and "musicality" can be considered similar words by a search engine. The stemming process normalizes each search token into a lexeme, a unit of lexical meaning that underlies a set of words that are related through inflection. The words "music", "musical" and "musicality" would convert to "music" when creating a search query.

Django provides a `SearchQuery` class to translate terms into a search query object. By default, the terms are passed through stemming algorithms, which helps you to obtain better matches.

The PostgreSQL search engine also removes stop words, such as "a", "the", "on", and "of". Stop words are a set of commonly used words in a language. They are removed when creating a search query because they appear too frequently to be relevant to searches. You can find the list of stop words used by PostgreSQL for the English language at `https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/english.stop`.

We also want to order results by relevancy. PostgreSQL provides a ranking function that orders results based on how often the query terms appear and how close together they are.

Edit the `views.py` file of the `blog` application and add the following imports:

```
from django.contrib.postgres.search import SearchVector, \
                                           SearchQuery, SearchRank
```

Then, edit the `post_search` view, as follows. New code is highlighted in bold:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            search_vector = SearchVector('title', 'body')
            search_query = SearchQuery(query)
            results = Post.published.annotate(
                search=search_vector,
                rank=SearchRank(search_vector, search_query)
            ).filter(search=search_query).order_by('-rank')

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

In the preceding code, we create a `SearchQuery` object, filter results by it, and use `SearchRank` to order the results by relevancy.

You can open `http://127.0.0.1:8000/blog/search/` in your browser and test different searches to test stemming and ranking. The following is an example of ranking by the number of occurrences of the word `django` in the title and body of the posts:

Figure 3.29: Search results for the term "django"

Stemming and removing stop words in different languages

We can set up `SearchVector` and `SearchQuery` to execute stemming and remove stop words in any language. We can pass a `config` attribute to `SearchVector` and `SearchQuery` to use a different search configuration. This allows us to use different language parsers and dictionaries. The following example executes stemming and removes stops in Spanish:

```
search_vector = SearchVector('title', 'body', config='spanish')
search_query = SearchQuery(query, config='spanish')
results = Post.published.annotate(
    search=search_vector,
    rank=SearchRank(search_vector, search_query)
).filter(search=search_query).order_by('-rank')
```

You can find the Spanish stop words dictionary used by PostgreSQL at `https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/spanish.stop`.

Weighting queries

We can boost specific vectors so that more weight is attributed to them when ordering results by relevancy. For example, we can use this to give more relevance to posts that are matched by title rather than by content.

Edit the `views.py` file of the `blog` application and modify the `post_search` view as follows. New code is highlighted in bold:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            search_vector = SearchVector('title', weight='A') + \
                            SearchVector('body', weight='B')
            search_query = SearchQuery(query)
            results = Post.published.annotate(
                search=search_vector,
                rank=SearchRank(search_vector, search_query)
            ).filter(rank__gte=0.3).order_by('-rank')
```

```
    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

In the preceding code, we apply different weights to the search vectors built using the `title` and `body` fields. The default weights are `D`, `C`, `B`, and `A`, and they refer to the numbers `0.1`, `0.2`, `0.4`, and `1.0`, respectively. We apply a weight of `1.0` to the `title` search vector (`A`) and a weight of `0.4` to the `body` vector (`B`). Title matches will prevail over body content matches. We filter the results to display only the ones with a rank higher than `0.3`.

Searching with trigram similarity

Another search approach is trigram similarity. A trigram is a group of three consecutive characters. You can measure the similarity of two strings by counting the number of trigrams that they share. This approach turns out to be very effective for measuring the similarity of words in many languages.

To use trigrams in PostgreSQL, you will need to install the `pg_trgm` extension first. Execute the following command in the shell prompt to connect to your database:

```
psql blog
```

Then, execute the following command to install the `pg_trgm` extension:

```
CREATE EXTENSION pg_trgm;
```

You will get the following output:

```
CREATE EXTENSION
```

Let's edit the view and modify it to search for trigrams.

Edit the `views.py` file of your `blog` application and add the following import:

```
from django.contrib.postgres.search import TrigramSimilarity
```

Then, modify the `post_search` view as follows. New code is highlighted in bold:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
```

```
            query = form.cleaned_data['query']
            results = Post.published.annotate(
                similarity=TrigramSimilarity('title', query),
            ).filter(similarity__gt=0.1).order_by('-similarity')

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

Open `http://127.0.0.1:8000/blog/search/` in your browser and test different searches for trigrams. The following example displays a hypothetical typo in the `django` term, showing search results for `yango`:

Figure 3.30: Search results for the term "yango"

We have added a powerful search engine to the blog application.

You can find more information about full-text search at `https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/`.

Additional resources

The following resources provide additional information related to the topics covered in this chapter:

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03`
- Django-taggit – `https://github.com/jazzband/django-taggit`

- Django-taggit ORM managers – https://django-taggit.readthedocs.io/en/latest/api.html
- Many-to-many relationships – https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/
- Django aggregation functions – https://docs.djangoproject.com/en/4.1/topics/db/aggregation/
- Built-in template tags and filters – https://docs.djangoproject.com/en/4.1/ref/templates/builtins/
- Writing custom template tags – https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/
- Markdown format reference – https://daringfireball.net/projects/markdown/basics
- Django Sitemap framework – https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/
- Django Sites framework – https://docs.djangoproject.com/en/4.1/ref/contrib/sites/
- Django syndication feed framework – https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/
- PostgreSQL downloads – https://www.postgresql.org/download/
- PostgreSQL full-text search capabilities – https://www.postgresql.org/docs/14/textsearch.html
- Django support for PostgreSQL full-text search – https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/

Summary

In this chapter, you implemented a tagging system by integrating a third-party application with your project. You generated post recommendations using complex QuerySets. You also learned how to create custom Django template tags and filters to provide templates with custom functionalities. You also created a sitemap for search engines to crawl your site and an RSS feed for users to subscribe to your blog. You then built a search engine for your blog using the full-text search engine of PostgreSQL.

In the next chapter, you will learn how to build a social website using the Django authentication framework and how to implement user account functionalities and custom user profiles.

4

Building a Social Website

In the preceding chapter, you learned how to implement a tagging system and how to recommend similar posts. You implemented custom template tags and filters. You also learned how to create sitemaps and feeds for your site, and you built a full-text search engine using PostgreSQL.

In this chapter, you will learn how to develop user account functionalities to create a social website, including user registration, password management, profile editing, and authentication. We will implement social features into this site in the next few chapters, to let users share images and interact with each other. Users will be able to bookmark any image on the internet and share it with other users. They will also be able to see activity on the platform from the users they follow and like/unlike the images shared by them.

This chapter will cover the following topics:

- Creating a login view
- Using the Django authentication framework
- Creating templates for Django login, logout, password change, and password reset views
- Extending the user model with a custom profile model
- Creating user registration views
- Configuring the project for media file uploads
- Using the messages framework
- Building a custom authentication backend
- Preventing users from using an existing email

Let's start by creating a new project.

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04`.

All Python packages used in this chapter are included in the `requirements.txt` file in the source code for the chapter. You can follow the instructions to install each Python package in the following sections, or you can install all requirements at once with the command `pip install -r requirements.txt`.

Creating a social website project

We are going to create a social application that will allow users to share images that they find on the internet. We will need to build the following elements for this project:

- An authentication system for users to register, log in, edit their profile, and change or reset their password
- A follow system to allow users to follow each other on the website
- Functionality to display shared images and a system for users to share images from any website
- An activity stream that allows users to see the content uploaded by the people that they follow

This chapter will address the first point on the list.

Starting the social website project

Open the terminal and use the following commands to create a virtual environment for your project:

```
mkdir env
python -m venv env/bookmarks
```

If you are using Linux or macOS, run the following command to activate your virtual environment:

```
source env/bookmarks/bin/activate
```

If you are using Windows, use the following command instead:

```
.\env\bookmarks\Scripts\activate
```

The shell prompt will display your active virtual environment, as follows:

```
(bookmarks)laptop:~ zenx$
```

Install Django in your virtual environment with the following command:

```
pip install Django~=4.1.0
```

Run the following command to create a new project:

```
django-admin startproject bookmarks
```

The initial project structure has been created. Use the following commands to get into your project directory and create a new application named `account`:

```
cd bookmarks/
django-admin startapp account
```

Remember that you should add the new application to your project by adding the application's name to the `INSTALLED_APPS` setting in the `settings.py` file.

Edit `settings.py` and add the following line highlighted in bold to the `INSTALLED_APPS` list before any of the other installed apps:

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

Django looks for templates in the application template directories by order of appearance in the `INSTALLED_APPS` setting. The `django.contrib.admin` app includes standard authentication templates that we will override in the `account` application. By placing the application first in the `INSTALLED_APPS` setting, we ensure that the custom authentication templates will be used by default instead of the authentication templates contained in `django.contrib.admin`.

Run the following command to sync the database with the models of the default applications included in the `INSTALLED_APPS` setting:

```
python manage.py migrate
```

You will see that all initial Django database migrations get applied. Next, we will build an authentication system into our project using the Django authentication framework.

Using the Django authentication framework

Django comes with a built-in authentication framework that can handle user authentication, sessions, permissions, and user groups. The authentication system includes views for common user actions such as logging in, logging out, password change, and password reset.

The authentication framework is located at `django.contrib.auth` and is used by other Django `contrib` packages. Remember that we already used the authentication framework in *Chapter 1*, *Building a Blog Application*, to create a superuser for the blog application to access the administration site.

When we create a new Django project using the `startproject` command, the authentication framework is included in the default settings of our project. It consists of the `django.contrib.auth` application and the following two middleware classes found in the `MIDDLEWARE` setting of our project:

- `AuthenticationMiddleware`: Associates users with requests using sessions
- `SessionMiddleware`: Handles the current session across requests

Middleware is classes with methods that are globally executed during the request or response phase. You will use middleware classes on several occasions throughout this book, and you will learn how to create custom middleware in *Chapter 17*, *Going Live*.

The authentication framework also includes the following models that are defined in `django.contrib.auth.models`:

- `User`: A user model with basic fields; the main fields of this model are `username`, `password`, `email`, `first_name`, `last_name`, and `is_active`
- `Group`: A group model to categorize users
- `Permission`: Flags for users or groups to perform certain actions

The framework also includes default authentication views and forms, which you will use later.

Creating a login view

We will start this section by using the Django authentication framework to allow users to log into the website. We will create a view that will perform the following actions to log in a user:

- Present the user with a login form
- Get the username and password provided by the user when they submit the form
- Authenticate the user against the data stored in the database
- Check whether the user is active
- Log the user into the website and start an authenticated session

We will start by creating the login form.

Create a new `forms.py` file in the `account` application directory and add the following lines to it:

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

This form will be used to authenticate users against the database. Note that you use the `PasswordInput` widget to render the `password` HTML element. This will include `type="password"` in the HTML so that the browser treats it as a password input.

Edit the `views.py` file of the `account` application and add the following code to it:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from .forms import LoginForm

def user_login(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
```

```
            cd = form.cleaned_data
            user = authenticate(request,
                                username=cd['username'],
                                password=cd['password'])
            if user is not None:
                if user.is_active:
                    login(request, user)
                    return HttpResponse('Authenticated successfully')
                else:
                    return HttpResponse('Disabled account')
            else:
                return HttpResponse('Invalid login')
    else:
        form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

This is what the basic login view does:

When the `user_login` view is called with a `GET` request, a new login form is instantiated with `form = LoginForm()`. The form is then passed to the template.

When the user submits the form via `POST`, the following actions are performed:

- The form is instantiated with the submitted data with `form = LoginForm(request.POST)`.
- The form is validated with `form.is_valid()`. If it is not valid, the form errors will be displayed later in the template (for example, if the user didn't fill in one of the fields).
- If the submitted data is valid, the user gets authenticated against the database using the `authenticate()` method. This method takes the `request` object, the `username`, and the `password` parameters and returns the `User` object if the user has been successfully authenticated, or `None` otherwise. If the user has not been successfully authenticated, a raw `HttpResponse` is returned with an **Invalid login** message.
- If the user is successfully authenticated, the user status is checked by accessing the `is_active` attribute. This is an attribute of Django's `User` model. If the user is not active, an `HttpResponse` is returned with a **Disabled account** message.
- If the user is active, the user is logged into the site. The user is set in the session by calling the `login()` method. An **Authenticated successfully** message is returned.

Note the difference between `authenticate()` and `login()`: `authenticate()` checks user credentials and returns a `User` object if they are correct; `login()` sets the user in the current session.

Now we will create a URL pattern for this view.

Create a new `urls.py` file in the `account` application directory and add the following code to it:

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

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

Edit the main `urls.py` file located in your `bookmarks` project directory, import `include`, and add the URL patterns of the `account` application, as follows. New code is highlighted in bold:

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

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

The login view can now be accessed by a URL.

Let's create a template for this view. Since there are no templates in the project yet, we will start by creating a base template that will be extended by the login template.

Create the following files and directories inside the `account` application directory:

```
templates/
    account/
        login.html
    base.html
```

Edit the `base.html` template and add the following code to it:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <span class="logo">Bookmarks</span>
  </div>
```

```
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
  </body>
  </html>
```

This will be the base template for the website. As you did in your previous project, include the CSS styles in the main template. You can find these static files in the code that comes with this chapter. Copy the `static/` directory of the `account` application from the chapter's source code to the same location in your project so that you can use the static files. You can find the directory's contents at `https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter04/bookmarks/account/static`.

The base template defines a `title` block and a `content` block that can be filled with content by the templates that extend from it.

Let's fill in the template for your login form.

Open the `account/login.html` template and add the following code to it:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  <p>Please, use the following form to log-in:</p>
  <form method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Log in"></p>
  </form>
{% endblock %}
```

This template includes the form that is instantiated in the view. Since your form will be submitted via `POST`, you will include the `{% csrf_token %}` template tag for **cross-site request forgery** (**CSRF**) protection. You learned about CSRF protection in *Chapter 2*, *Enhancing Your Blog with Advanced Features*.

There are no users in the database yet. You will need to create a superuser first to access the administration site to manage other users.

Execute the following command in the shell prompt:

```
python manage.py createsuperuser
```

You will see the following output. Enter your desired username, email, and password, as follows:

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
```

Then you will see the following success message:

```
Superuser created successfully.
```

Run the development server using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/` in your browser. Access the administration site using the credentials of the user you just created. You will see the Django administration site, including the `User` and `Group` models of the Django authentication framework.

It will look as follows:

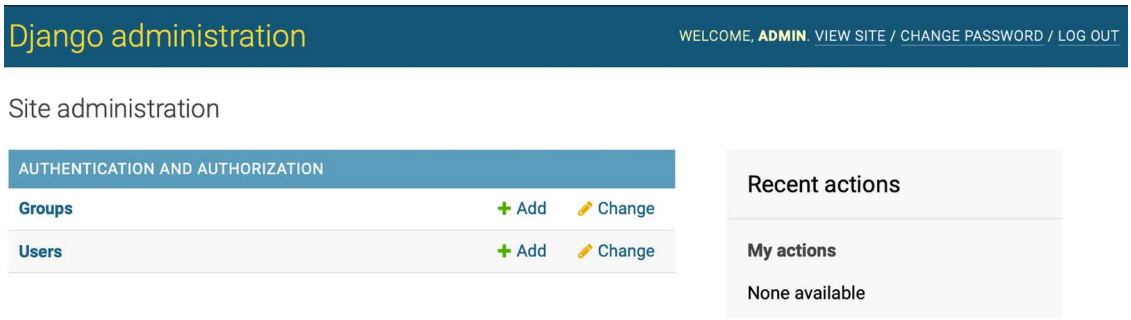


Figure 4.1: The Django administration site index page including Users and Groups

In the **Users** row, click on the **Add** link.

Create a new user using the administration site as follows:

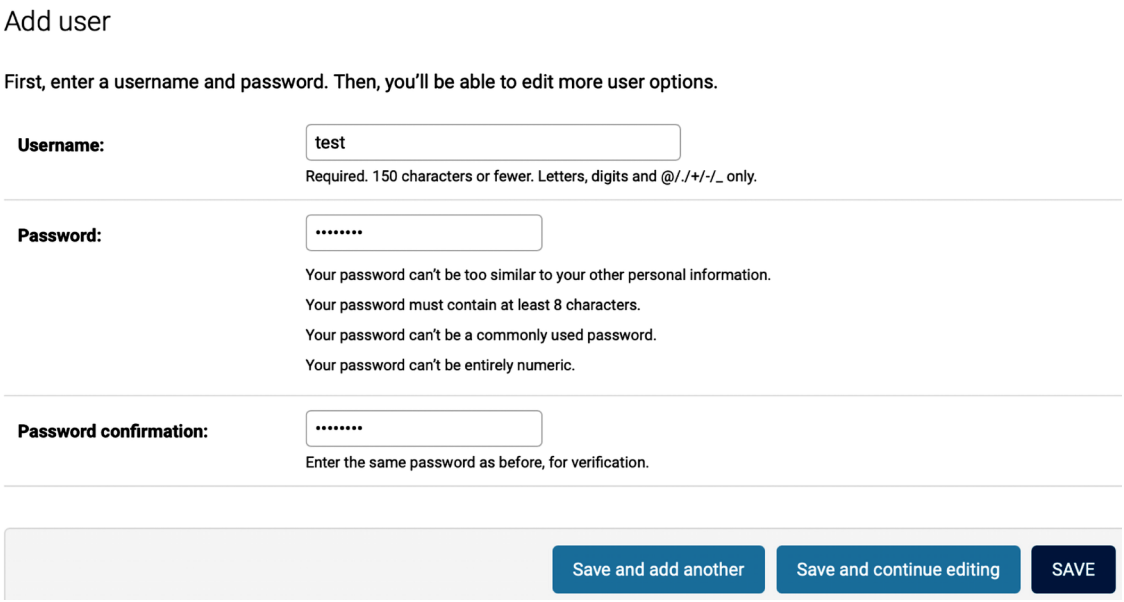


Figure 4.2: The Add user form on the Django administration site

Enter the user details and click on the **SAVE** button to save the new user in the database.

Then, in **Personal info**, fill in the **First name**, **Last name**, and **Email address** fields as follows and click on the **Save** button to save the changes:

Figure 4.3: The user editing form in the Django administration site

Open `http://127.0.0.1:8000/account/login/` in your browser. You should see the rendered template, including the login form:

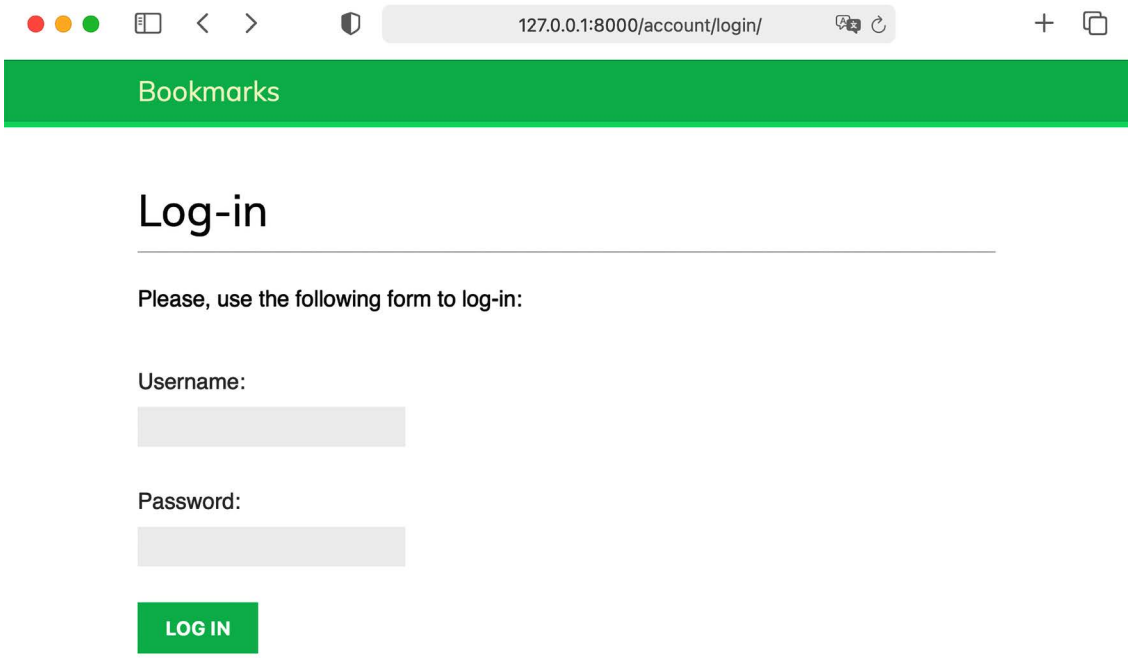


Figure 4.4: The user Log-in page

Enter invalid credentials and submit the form. You should get the following **Invalid login** response:

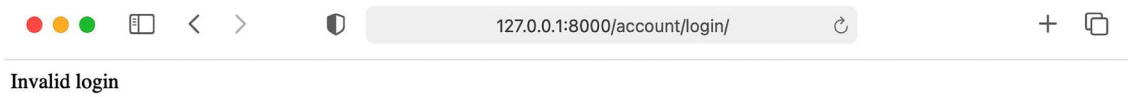


Figure 4.5: The invalid login plain text response

Enter valid credentials; you will get the following **Authenticated successfully** response:

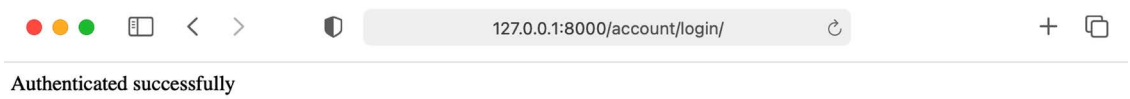


Figure 4.6: The successful authentication plain text response

You have learned how to authenticate users and create your own authentication view. You can build your own auth views but Django ships with ready-to-use authentication views that you can leverage.

Using Django authentication views

Django includes several forms and views in the authentication framework that you can use right away. The login view we have created is a good exercise to understand the process of user authentication in Django. However, you can use the default Django authentication views in most cases.

Django provides the following class-based views to deal with authentication. All of them are located in `django.contrib.auth.views`:

- `LoginView`: Handles a login form and logs in a user
- `LogoutView`: Logs out a user

Django provides the following views to handle password changes:

- `PasswordChangeView`: Handles a form to change the user's password
- `PasswordChangeDoneView`: The success view that the user is redirected to after a successful password change

Django also includes the following views to allow users to reset their password:

- `PasswordResetView`: Allows users to reset their password. It generates a one-time-use link with a token and sends it to a user's email account
- `PasswordResetDoneView`: Tells users that an email—including a link to reset their password—has been sent to them
- `PasswordResetConfirmView`: Allows users to set a new password
- `PasswordResetCompleteView`: The success view that the user is redirected to after successfully resetting their password

These views can save you a lot of time when building any web application with user accounts. The views use default values that can be overridden, such as the location of the template to be rendered, or the form to be used by the view.

You can get more information about the built-in authentication views at `https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views`.

Login and logout views

Edit the `urls.py` file of the `account` application and add the code highlighted in bold:

```
from django.urls import path
from django.contrib.auth import views as auth_views
from . import views

urlpatterns = [
    # previous login url
    # path('login/', views.user_login, name='login'),
```

```
    # login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

In the preceding code, we have commented out the URL pattern for the `user_login` view that we created previously. We'll now use the `LoginView` view of Django's authentication framework. We have also added a URL pattern for the `LogoutView` view.

Create a new directory inside the `templates/` directory of the `account` application and name it `registration`. This is the default path where the Django authentication views expect your authentication templates to be.

The `django.contrib.admin` module includes authentication templates that are used for the administration site, like the login template. By placing the `account` application at the top of the `INSTALLED_APPS` setting when configuring the project, we ensured that Django would use our authentication templates instead of the ones defined in any other application.

Create a new file inside the `templates/registration/` directory, name it `login.html`, and add the following code to it:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  {% if form.errors %}
    <p>
      Your username and password didn't match.
      Please try again.
    </p>
  {% else %}
    <p>Please, use the following form to log-in:</p>
  {% endif %}
  <div class="login-form">
    <form action="{% url 'login' %}" method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="hidden" name="next" value="{{ next }}" />
      <p><input type="submit" value="Log-in"></p>
    </form>
  </div>
{% endblock %}
```

This login template is quite similar to the one we created before. Django uses the `AuthenticationForm` form located at `django.contrib.auth.forms` by default. This form tries to authenticate the user and raises a validation error if the login is unsuccessful. We use `{% if form.errors %}` in the template to check whether the credentials provided are wrong.

We have added a hidden HTML `<input>` element to submit the value of a variable called `next`. This variable is provided to the login view if you pass a parameter named `next` to the request, for example, by accessing `http://127.0.0.1:8000/account/login/?next=/account/`.

The `next` parameter has to be a URL. If this parameter is given, the Django login view will redirect the user to the given URL after a successful login.

Now, create a `logged_out.html` template inside the `templates/registration/` directory and make it look like this:

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

{% block title %}Logged out{% endblock %}

{% block content %}
  <h1>Logged out</h1>
  <p>
    You have been successfully logged out.
    You can <a href="{% url "login" %}">log-in again</a>.
  </p>
{% endblock %}
```

This is the template that Django will display after the user logs out.

We have added the URL patterns and templates for the login and logout views. Users can now log in and out using Django's authentication views.

Now, we will create a new view to display a dashboard when users log into their accounts.

Edit the `views.py` file of the `account` application and add the following code to it:

```
from django.contrib.auth.decorators import login_required

@login_required
def dashboard(request):
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard'})
```

We have created the `dashboard` view, and we have applied to it the `login_required` decorator of the authentication framework. The `login_required` decorator checks whether the current user is authenticated.

If the user is authenticated, it executes the decorated view; if the user is not authenticated, it redirects the user to the login URL with the originally requested URL as a `GET` parameter named `next`.

By doing this, the login view redirects users to the URL that they were trying to access after they successfully log in. Remember that we added a hidden `<input>` HTML element named `next` in the login template for this purpose.

We have also defined a `section` variable. We will use this variable to highlight the current section in the main menu of the site.

Next, we need to create a template for the dashboard view.

Create a new file inside the `templates/account/` directory and name it `dashboard.html`. Add the following code to it:

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

{% block title %}Dashboard{% endblock %}

{% block content %}
  <h1>Dashboard</h1>
  <p>Welcome to your dashboard.</p>
{% endblock %}
```

Edit the `urls.py` file of the `account` application and add the following URL pattern for the view. The new code is highlighted in bold:

```
urlpatterns = [
    # previous login url
    # path('login/', views.user_login, name='login'),

    # login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    path('', views.dashboard, name='dashboard'),
]
```

Edit the `settings.py` file of the project and add the following code to it:

```
LOGIN_REDIRECT_URL = 'dashboard'
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
```

We have defined the following settings:

- `LOGIN_REDIRECT_URL`: Tells Django which URL to redirect the user to after a successful login if no `next` parameter is present in the request
- `LOGIN_URL`: The URL to redirect the user to log in (for example, views using the `login_required` decorator)
- `LOGOUT_URL`: The URL to redirect the user to log out

We have used the names of the URLs that we previously defined with the `name` attribute of the `path()` function in the URL patterns. Hardcoded URLs instead of URL names can also be used for these settings.

Let's summarize what we have done so far:

- We have added the built-in Django authentication login and logout views to the project.
- We have created custom templates for both views and defined a simple dashboard view to redirect users after they log in.
- Finally, we have added settings for Django to use these URLs by default.

Now, we will add login and logout links to the base template. In order to do this, we have to determine whether the current user is logged in or not in order to display the appropriate link for each case. The current user is set in the `HttpRequest` object by the authentication middleware. You can access it with `request.user`. You will find a `User` object in the request even if the user is not authenticated. A non-authenticated user is set in the request as an instance of `AnonymousUser`. The best way to check whether the current user is authenticated is by accessing the read-only attribute `is_authenticated`.

Edit the `templates/base.html` template by adding the following lines highlighted in bold:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <span class="logo">Bookmarks</span>
    {% if request.user.is_authenticated %}
      <ul class="menu">
        <li {% if section == "dashboard" %}class="selected"{% endif %}>
          <a href="{% url "dashboard" %}">My dashboard</a>
        </li>
        <li {% if section == "images" %}class="selected"{% endif %}>
          <a href="#">Images</a>
        </li>
```

```
        <li {% if section == "people" %}class="selected"{% endif %}>
          <a href="#">People</a>
        </li>
      </ul>
    {% endif %}
    <span class="user">
      {% if request.user.is_authenticated %}
        Hello {{ request.user.first_name|default:request.user.username }},
        <a href="{% url "logout" %}">Logout</a>
      {% else %}
        <a href="{% url "login" %}">Log-in</a>
      {% endif %}
    </span>
  </div>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
</body>
</html>
```

The site's menu is only displayed to authenticated users. The `section` variable is checked to add a `selected` class attribute to the menu `<li>` list item of the current section. By doing so, the menu item that corresponds to the current section will be highlighted using CSS. The user's first name and a link to log out are displayed if the user is authenticated; a link to log in is displayed otherwise. If the user's name is empty, the username is displayed instead by using `request.user.first_name|default:request.user.username`.

Open `http://127.0.0.1:8000/account/login/` in your browser. You should see the **Log-in** page. Enter a valid username and password and click on the **Log-in** button. You should see the following screen:

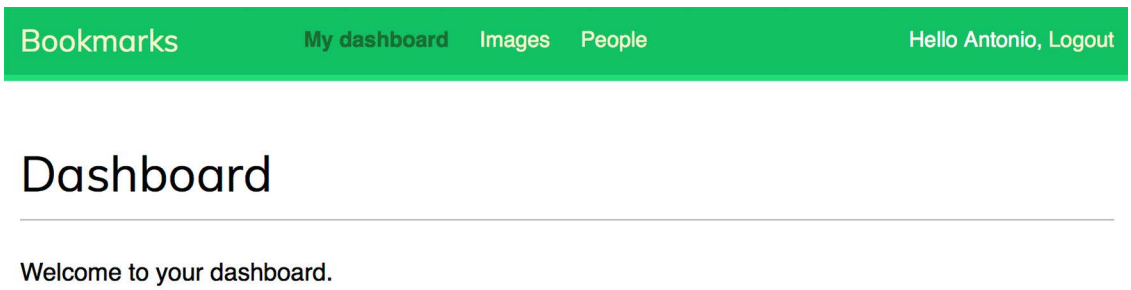


Figure 4.7: The Dashboard page

The **My dashboard** menu item is highlighted with CSS because it has a `selected` class. Since the user is authenticated, the first name of the user is displayed on the right side of the header. Click on the **Logout** link. You should see the following page:

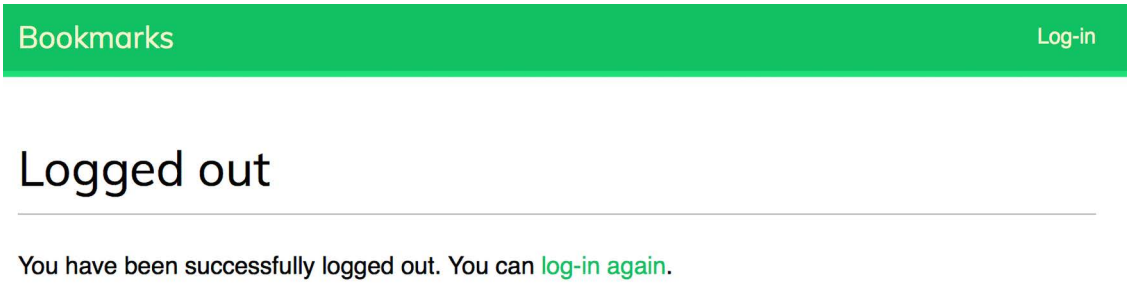


Figure 4.8: The Logged out page

On this page, you can see that the user is logged out, and, therefore, the menu of the website is not displayed. The link displayed on the right side of the header is now **Log-in**.

If you see the **Logged out** page of the Django administration site instead of your own **Logged out** page, check the `INSTALLED_APPS` setting of your project and make sure that `django.contrib.admin` comes after the `account` application. Both applications contain logged-out templates located in the same relative path. The Django template loader will go through the different applications in the `INSTALLED_APPS` list and use the first template it finds.

Change password views

We need users to be able to change their password after they log into the site. We will integrate the Django authentication views for changing passwords.

Open the `urls.py` file of the `account` application and add the following URL patterns highlighted in bold:

```
urlpatterns = [
    # previous login url
    # path('login/', views.user_login, name='login'),

    # login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # change password urls
    path('password-change/',
         auth_views.PasswordChangeView.as_view(),
         name='password_change'),
    path('password-change/done/',
```

```
            auth_views.PasswordChangeDoneView.as_view(),
            name='password_change_done'),

    path('', views.dashboard, name='dashboard'),
]
```

The `PasswordChangeView` view will handle the form to change the password, and the `PasswordChangeDoneView` view will display a success message after the user has successfully changed their password. Let's create a template for each view.

Add a new file inside the `templates/registration/` directory of the `account` application and name it `password_change_form.html`. Add the following code to it:

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

{% block title %}Change your password{% endblock %}

{% block content %}
  <h1>Change your password</h1>
  <p>Use the form below to change your password.</p>
  <form method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Change"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

The `password_change_form.html` template includes the form to change the password.

Now create another file in the same directory and name it `password_change_done.html`. Add the following code to it:

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

{% block title %}Password changed{% endblock %}

{% block content %}
  <h1>Password changed</h1>
  <p>Your password has been successfully changed.</p>
{% endblock %}
```

The `password_change_done.html` template only contains the success message to be displayed when the user has successfully changed their password.

Open `http://127.0.0.1:8000/account/password-change/` in your browser. If you are not logged in, the browser will redirect you to the **Log-in** page. After you are successfully authenticated, you will see the following change password page:

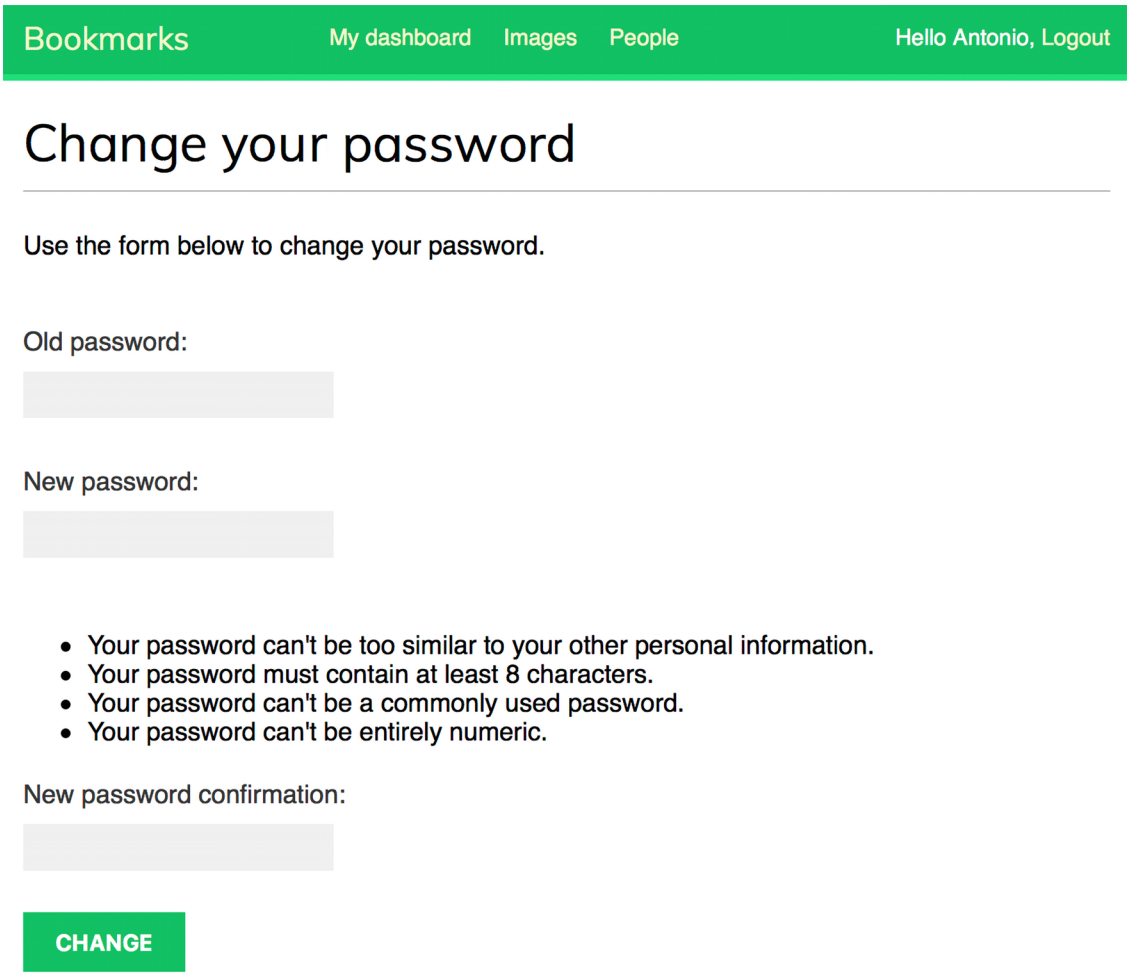


Figure 4.9: The change password form