Tom Aratyn

Building Django 2.0 Web Applications

Create enterprise-grade, scalable Python web applications easily with Django 2.0

Packt>

Building Django 2.0 Web Applications

Create enterprise-grade, scalable Python web applications easily with Django 2.0

Tom Aratyn

BIRMINGHAM - MUMBAI

Building Django 2.0 Web Applications

Commissioning Editor: Amarabha Banerjee
Acquisition Editor: Noyonika Das
Content Development Editor: Gauri Pradhan
Technical Editor: Rutuja Vaze
Copy Editor: Dhanya Baburaj
Project Coordinator: Sheejal Shah
Proofreader: Safis Editing
Indexer: Aishwarya Gangawane
Graphics: Jason Monteiro
Production Coordinator: Shraddha Falebhai

First published: April 2018

Production reference: 1250418

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

ISBN 978-1-78728-621-4

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

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

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

Contributors

About the author

Tom Aratyn is a software developer and the founder of the Boulevard Platform. He has a decade of experience developing web apps for companies of all sizes (from boutiques to large start-ups, such as Snapchat). He loves solving problems using his server-side and client-side development skills and helping other developers grow.

I want to thank the many people who made this book possible. Thanks mom! Thanks to my friends who helped keep me grounded through it all. Particular thanks to my editor, Gauri Pradhan, who helped me so much over the many months. Thank you to the reviewers Andrei Kulakov and Dan Noble for helping improve the book. My thanks also to the many other folks on the Packt team, including Dhanya Baburaj, Rutuja Vaze, Noyonika Das, and everyone else!

About the reviewers

Andrei Kulakov lives in New York and has worked in the software industry for 10 years, including many projects in genetic research, linguistics, hardware systems, healthcare and machine learning. In his spare time, Andrei can often be found practicing hand-balancing in one of the city parks.

Dan Noble is an accomplished full-stack web developer, data engineer, and author with more than 10 years of development experience. He enjoys working with a variety of programming languages and software frameworks, particularly Python, Elasticsearch, and JavaScript.

Dan currently works on geospatial web applications and data processing systems. He has been a user and an advocate of Django and Elasticsearch since 2009. He is the author of the book Monitoring Elasticsearch and a technical reviewer for The *Elasticsearch Cookbook, Second Edition*, by Alberto Paro, by Packt Publishing.

Packt is searching for authors like you

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

Table of Contents

Preface

Who doesn't have an idea for the next great app or service they want to launch? However, most apps, services, and websites ultimately rely on a server being able to accept requests and then create, read, update, and delete records based on those requests. Django makes it easy to build and launch websites, services, and backends for your great idea. However, despite the history of being used at large-scale successful start-ups and enterprises, it can be difficult to gather all the resources necessary to actually take an idea from empty directory to running production server.

Over the course of three projects, *Building Django Web Applications* guides you from an empty directory to creating full-fledged apps to replicate the core functionality of some of the web's most popular web apps. In Part 1, you'll make your own online movie database. In Part 2, you'll make a website letting users ask and answer questions. In Part 3, you'll make a web app to manage mailing lists and send emails. All three projects culminate in your deploying the project to a server so that you can see your ideas come to life. Between starting each project and deploying it, we'll cover important practical concepts such as how to build APIs, secure your project, add search using Elasticsearch, use caching, and offload tasks to worker process to help your project scales.

Building Django Web Applications is for developers who already know some basics of Python, but want to take their skills to the next level. Basic understanding of HTML and CSS is also recommended, as these languages will be mentioned but are not the focus of the book.

After reading this book, you'll be familiar with everything it takes to launch an amazing web app using Django.

Who this book is for

This book is for developers who are familiar with Python. Readers should know how to run commands in Bash shell. Some basic HTML and CSS knowledge is assumed. Finally, the reader should be able to connect to a PostgreSQL database on their own.

What this book covers

`Chapter 1`, *Building MyMDB*, covers starting a Django project and the core MyMDB Django app. You will create the core models, views, and templates. You will create URLConfs to help Django route requests to your views. By the end of this chapter, you will have a tested Django project that you can access using your web browser.

`Chapter 2`, Adding Users to MyMDB, covers adding user registration and authentication. With users being able to register, log in, and log out, you will accept and display votes on movies. Finally, you'll write aggregate queries using Django's QuerySet API to score each movie.

`Chapter 3`, *Posters, Headshots, and Security*, covers securely accepting and storing files from your users. You'll learn about the top web application security issues, as listed in the OWASP Top Ten, and how Django mitigates those issues.

`Chapter 4`, *Caching in on the Top 10*, covers how to optimize Django projects. You'll learn how to measure what needs optimization. Finally, you'll learn about the different caching strategies Django makes available and when to use them.

`Chapter 5`, *Deploying with Docker*, covers how to deploy Django using Nginx and uWSGI in a Docker container. You'll learn how to store uploaded files in S3 to protect the user. Finally, you'll run your Docker container on a Linux server in the Amazon Web Services cloud.

`Chapter 6`, *Starting Answerly*, covers creating the models, views, templates, and apps for the Answerly project. You'll learn how to use Django's built-in date views to show a list of questions asked each day. You'll also learn how to split large templates into more manageable components.

`Chapter 7`, *Searching for Questions with Elasticsearch*, covers working with Elasticsearch to let users search our questions. You will learn how to create a service that avoid coupling external services to your models or views. You will also learn how to automatically load and update model data in an external service.

`Chapter 8`, *Testing Answerly*, covers how to test a Django project. You will learn how to measure code coverage in a Django project and how to easily generate test data. You will also learn how to write different types of tests from unit tests to live server tests with a working browser.

`Chapter 9`, *Deploying Answerly*, covers how to deploy a Django project on a Linux server with Apache and mod_wsgi. You'll also learn how to treat your Django project like a twelve-factor app to keep it easy to scale.

`Chapter 10`, *Starting Mail Ape*, covers creating the models, views, templates, and apps for the Mail Ape project. You'll learn how to use alternate fields for non-sequential primary keys.

`Chapter 11`, *Sending Emails*, covers how to use Django's email functionality. You'll also learn how to use Celery to process tasks outside of the Django request/response cycle and how to test code that relies on Celery tasks.

`Chapter 12`, *Building an API*, covers how to create an API using the **Django REST Framework** (**DRF**). You'll learn how DRF lets you quickly build an API from your Django models without repeating a lot of unnecessary code. You will also learn how to access and test your API.

`Chapter 13`, *Deploying Mail Ape*, covers how to deploy a Django app into the Amazon Web Services cloud. You'll learn how to make an **Amazon Machine Image** (**AMI**) a part of a release. Then, you'll create a CloudFormation template to declare your infrastructure and servers as code. You'll take a look at how to use AWS to horizontally scale your system to run multiple web workers. Finally, you'll bring it all online using the AWS command-line interface.

To get the most out of this book

To get the most out of this book book you should:

1. Have some familiarity with Python and have Python3.6+ installed
2. Be able to install Docker or other new software on your computer
3. Know how to connect to a Postgres server from your computer
4. Have access to a Bash shell

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Building-Django-2.0-Web-Applications`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

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. Here is an example: "It also offers a `create()` method for creating and saving an instance."

A block of code is set as follows:

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

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

```
DATABASES = {
  'default': {
     'ENGINE': 'django.db.backends.sqlite3',
     'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
  }
}
```

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

```
$ pip install -r requirements.dev.txt
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Clicking on **MOVIES** will show us a list of movies."

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 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 would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 `authors.packtpub.com`.

Reviews

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

For more information about Packt, please visit `packtpub.com`.

1
Starting MyMDB

The first project we will build is a basic **Internet Movie Database** (**IMDB**) clone called **My Movie Database (MyMDB)** written in Django 2.0 that we will deploy using Docker. Our IMDB clone will have the following two types of users: users and administrators. The users will be able to rate movies, add images from movies, and view movies and cast. The administrators will be able to add movies, actors, writers, and directors.

In this chapter, we'll do the following things:

- Create our new Django project MyMDB, an IMDB clone
- Make a Django app and create our first models, views, and templates
- Learn about and use a variety of fields in our models and create relationships across models

The code for this project is available online at `https://github.com/tomaratyn/MyMDB`.

By the end, we'll be able to add movies, people, and roles into our project and let users view them in easy-to-customize HTML templates.

Starting My Movie Database (MyMDB)

First, let's make a directory for our project:

```
$ mkdir MyMDB
$ cd MyMDB
```

All our future commands and paths will be relative to this project directory.

Starting the project

A Django project is composed of multiple Django apps. A Django app can come from many different places:

- Django itself (for example, `django.contrib.admin`, the admin backend app)
- Installed Python packages (for example, `django-rest-framework`, a framework for creating REST APIs from Django models)
- Written as part of the project (the code we'll be writing)

Usually, a project uses a mix of all of the preceding three options.

Installing Django

We'll install Django using `pip`, Python's preferred package manager and track which packages we install in a `requirements.dev.txt` file:

```
django<2.1
psycopg2<2.8
```

Now, let's install the packages:

```
$ pip install -r requirements.dev.txt
```

Creating the project

With Django installed, we have the `django-admin` command-line tool with which we can generate our project:

```
$ django-admin startproject config
$ tree config/
config/
├── config
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

The parent of the `settings.py` file is called `config` because we named our project `config` instead of `mymdb`. However, letting that top-level directory continue to be called `config` is confusing, so let's just rename it `django` (a project may grow to contain lots of different types of code; calling the parent of the Django code `django`, again, makes it clear):

```
$ mv config django
$ tree .
.
├── django
│   ├── config
│   │   ├── __init__.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   └── manage.py
└── requirements.dev.txt

2 directories, 6 files
```

Let's take a closer look at some of these files:

- `settings.py`: This is where Django stores all the configuration for your app by default. In the absence of a `DJANGO_SETTINGS` environment variable, this is where Django looks for settings by default.
- `urls.py`: This is the root `URLConf` for the entire project. Every request that your web app gets will get routed to the first view that matches a path inside this file (or a file `urls.py` reference).
- `wsgi.py`: **Web Server Gateway Interface** (**WSGI**) is the interface between Python and a web server. You won't touch this file very much, but it's how your web server and your Python code know how to talk to each other. We'll reference it in `Chapter 5`, *Deploying with Docker*.
- `manage.py`: This is the command center for making non-code changes. Whether it's creating a database migration, running tests, or starting the development server, we will use this file often.

Note what's missing is that the `django` directory is not a Python module. There's no `__init__.py` file in there, and there should *not* be. If you add one, many things will break because we want the Django apps we add to be top-level Python modules.

Configuring database settings

By default, Django creates a project that will use SQLite, but that's not usable for production, so we'll follow the best practice of using the same database in development as in production.

Let's open up `django/config/settings.py` and update it to use our Postgres server. Find the line in `settings.py` that starts with `DATABASES`. By default, it will look like this:

```
DATABASES = {
  'default': {
     'ENGINE': 'django.db.backends.sqlite3',
     'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
  }
}
```

To use Postgres, change the preceding code to the following one:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mymdb',
        'USER': 'mymdb',
        'PASSWORD': 'development',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

Most of this will seem familiar if you've connected to a database before, but let's review:

- `DATABASES = {`: This constant is a dictionary of database connection information and is required by Django. You can have multiple connections to different databases, but, most of the time, you will just need an entry called `default`.
- `'default': {`: This is the default database connection configuration. You should always have a `default` set of connections settings. Unless you specify otherwise (and, in this book, we won't), this is the connection you'll be using.
- `'ENGINE': 'django.db.backends.postgresql '`: This tells Django to use the Postgres backend. This in turn uses `psycopg2`, Python's Postgres library.
- `'NAME': 'mymdb',`: The name of the database you want to connect to.

- 'USER': 'mymdb',: The username for your connection.
- 'PASSWORD': 'development',: The password for your database user.
- 'HOST': '127.0.0.1',: The address of the database server you want to connect to.
- 'PORT': '5432',: The port you want to connect to.

The core app

Django apps follow a **Model View Template** (**MVT**) pattern; in this pattern, we will note the following things:

- **Models** are responsible for saving and retrieving data from the database
- **Views** are responsible for processing HTTP Requests, initiating operations on Models, and returning HTTP responses
- **Templates** are responsible for the look of the response body

There's no limit on how many apps you can have in your Django project. Ideally, each app should have a tightly scoped and self-contained functionality like any other Python module, but at the beginning of a project, it can be hard to know where the complexity will lie. That's why I find it useful to start off with a `core` app. Then, when I notice clusters of complexity around particular topics (let's say, in our project, actors could become unexpectedly complex if we're getting traction there), then we can refactor that into its own tightly scoped app. Other times, it's clear that a site has self-contained components (for example, an admin backend), and it's easy to start off with multiple apps.

Making the core app

To make a new Django app, we first have to use `manage.py` to create the app and then add it to the list of `INSTALLED_APPS`:

```
$ cd django
$ python manage.py startapp core
$ ls
config      core        manage.py
$tree core
core
├──  472; __init__.py
├─── admin.py
├─── apps.py
```

```
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py

1 directory, 7 files
```

Let's take a closer look at what's inside of the core:

- `core/__init__.py`: The core is not just a directory, but also a Python module.
- `admin.py`: This is where we will register our models with the built-in admin backend. We'll describe that in the *Movie Admin* section.
- `apps.py`: Most of the time, you'll leave this alone. This is where you would put any code that needs to run when registering your application, which is useful if you're making a reusable Django app (for example, a package you want to upload to PyPi).
- `migrations`: This is a Python module with database migrations. Database migrations describe how to *migrate* the database from one known state to another. With Django, if you add a model, you can just generate and run a migration using `manage.py`, which you can see later in this chapter in the *Migrating the database* section.
- `models.py`: This is for models.
- `tests.py`: This is for tests.
- `views.py`: This is for views.

Installing our app

Now that our core app exists, let's make Django aware of it by adding it to the list of installed apps in `settings.py` file. Your `settings.py` should have a line that looks like this:

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

`INSTALLED_APPS` is a list of Python paths to Python modules that are Django apps. We already have apps installed to solve common problems, such as managing static files, sessions, and authentication and an admin backend because of Django's Batteries Included philosophy.

Let's add our `core` app to the top of that list:

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

Adding our first model – Movie

Now we can add our first model, that is, Movie.

A Django model is a class that is derived from `Model` and has one or more `Fields`. In database terms, a `Model` class corresponds to a database table, `Field` classes correspond to columns, and instances of a `Model` correspond to rows. Using an ORM like Django's, let's take advantage of Python and Django to write expressive classes instead of DB writing our models once in Python and again in SQL.

Let's edit `django/core/models.py` to add a `Movie` model:

```
from django.db import models

class Movie(models.Model):
    NOT_RATED = 0
    RATED_G = 1
    RATED_PG = 2
    RATED_R = 3
    RATINGS = (
        (NOT_RATED, 'NR - Not Rated'),
        (RATED_G,
         'G - General Audiences'),
        (RATED_PG,
         'PG - Parental Guidance '
         'Suggested'),
        (RATED_R, 'R - Restricted'),
```

```
)

title = models.CharField(
    max_length=140)
plot = models.TextField()
year = models.PositiveIntegerField()
rating = models.IntegerField(
    choices=RATINGS,
    default=NOT_RATED)
runtime = \
    models.PositiveIntegerField()
website = models.URLField(
    blank=True)

def __str__(self):
    return '{} ({})'.format(
        self.title, self.year)
```

`Movie` is derived from `models.Model`, which is the base class for all Django models. Next, there's a series of constants that describe ratings; we'll take a look at that when we look at the `rating` field, but first let's look at the other fields:

- `title = models.CharField(max_length=140)`: This will become a `varchar` column with a length of 140. Databases generally require a maximum size for `varchar` columns, so Django does too.
- `plot = models.TextField()`: This will become a `text` column in our database, which has no maximum length requirement. This makes it more appropriate for a field that can have a paragraph (or even pages) of text.
- `year = models.PositiveIntegerField()`: This will become an `integer` column, and Django will validate the value before saving it to ensure that it is `0` or higher when you save it.
- `rating = models.IntegerField(choices=RATINGS, default=NOT_RATED)`: This is a more complicated field. Django will know that this is going to be an `integer` column. The optional argument `choices` (which is available for all `Fields`, not just `IntegerField`) takes an iterable (list or tuple) of value/display pairs. The first element in the pair is a valid value that can be stored in the database and the second is a human-friendly version of the value. Django will also add an instance method to our model called `get_rating_display()`, which will return the matching second element for the value stored in our model. Anything that doesn't match one of the values in `choices` will be a `ValidationError` on save. The `default` argument provides a default value if one is not provided when creating the model.

- `runtime = models.PositiveIntegerField()`: This is the same as the `year` field.
- `website = models.URLField(blank=True)`: Most databases don't have a native URL column type, but data-driven web apps often need to store them. A `URLField` is a `varchar(200)` field by default (this can be set by providing a `max_length` argument). `URLField` also comes with validation, checking whether its value is a valid web (`http`/`https`/`ftp`/`ftps`) URL. The `blank` argument is used by the `admin` app to know whether to require a value (it does not affect the database).

Our model also has a `__str__(self)` method, which is a best practice that helps Django convert the model to a string. Django does this in the administrative UI and in our own debugging.

Django's ORM automatically adds an autoincrementing `id` column, so we don't have to repeat that on all our models. It's a simple example of Django's **Don't Repeat Yourself (DRY)** philosophy. We'll take a look at more examples as we go along.

Migrating the database

Now that we have a model, we will need to create a table in our database that matches it. We will use Django to generate a migration for us and then run the migration to create a table for our movie model.

While Django can create and run migrations for our Django apps, it will not create the database and database user for our Django project. To create the database and user, we have to connect to the server using an administrator's account. Once we've connected we can create the database and user by executing the following SQL:

```
CREATE DATABASE mymdb;
CREATE USER mymdb;
GRANT ALL ON DATABASE mymdb to "mymdb";
ALTER USER mymdb PASSWORD 'development';
ALTER USER mymdb CREATEDB;
```

The above SQL statements will create the database and user for our Django project. The `GRANT` statement ensures that our mymdb user will have access to the database. Then, we set a password on the `mymdb` user (make sure it's the same as in your `settings.py` file). Finally, we give the `mymdb` user permission to create new databases, which will be used by Django to create a test database when running tests.

To generate a migration for our app, we'll need to tell `manage.py` file to do as follows:

```
$ cd django
$ python manage.py makemigrations core
Migrations for 'core':
  core/migrations/0001_initial.py
    - Create model Movie
```

A `migration` is a Python file in our Django app that describes how to change the database into a desired state. Django migrations are not tied to a particular database system (the same migrations will work across supported databases, unless *we* add database-specific code). Django generates migration files that use Django's migrations API, which we won't be looking at in this book, but it's useful to know that it exists.

Remember that it's *apps* not *projects* that have migrations (since it's *apps* that have models).

Next, we tell `manage.py` to migrate our app:

```
$ python manage.py migrate core
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0001_initial... OK
```

Now, our table exists in our database:

```
$ python manage.py dbshell
psql (9.6.1, server 9.6.3)
Type "help" for help.

mymdb=> \dt
             List of relations
 Schema |       Name         | Type  | Owner
--------+--------------------+-------+-------
 public | core_movie         | table | mymdb
 public | django_migrations | table | mymdb
(2 rows)

mymdb=> \q
```

We can see that our database has two tables. The default naming scheme for Django's model's tables is `<app_name>_<model_name>`. We can tell `core_movie` is the table for the `Movie` model from the `core` app. `django_migrations` is for Django's internal use to track the migrations that have been applied. Altering the `django_migrations` table directly instead of using `manage.py` is a bad idea, which will lead to problems when you try to apply or roll back migrations.

The migration commands can also run without specifying an app, in which case it will run on all the apps. Let's run the `migrate` command without an app:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, core, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying 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 sessions.0001_initial... OK
```

This creates tables to keep track of users, sessions, permissions, and the administrative backend.

Creating our first movie

Like Python, Django offers an interactive REPL to try things out. The Django shell is fully connected to the database, so we can create, query, update, and delete models from the shell:

```
$ cd django
$ python manage.py shell
Python 3.4.6 (default, Aug  4 2017, 15:21:32)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from core.models import Movie
```

```
>>> sleuth = Movie.objects.create(
... title='Sleuth',
... plot='An snobbish writer who loves games'
... ' invites his wife\'s lover for a battle of wits.',
... year=1972,
... runtime=138,
... )
>>> sleuth.id
1
>>> sleuth.get_rating_display()
'NR - Not Rated'
```

In the preceding Django shell session, note that there are a number of attributes of `Movie` that we didn't create:

- `objects` is the model's default manager. Managers are an interface for querying the model's table. It also offers a `create()` method for creating and saving an instance. Every model must have at least one manager, and Django offers a default manager. It's often advisable to create a custom manager; we'll see that later in the *Adding Person and model relationships* section.
- `id` is the primary key of the row for this instance. As mentioned in the preceding step, Django creates it automatically.
- `get_rating_display()` is a method that Django added because the `rating` field was given a tuple of `choices`. We didn't have to provide `rating` with a value in our `create()` call because the `rating` field has a `default` value (`0`). The `get_rating_display()` method looks up the value and returns the corresponding display value. Django will generate a method like this for each `Field` attribute with a `choices` argument.

Next, let's create a backend for managing movies using the Django Admin app.

Creating movie admin

Being able to quickly generate a backend UI lets users to start building the content of the project while the rest of the project is still in development. It's a nice feature that helps parallelize progress and avoid a repetitious and boring task (read/update views share a lot of functionalities). Providing this functionality out of the box is another example of Django's Batteries Included philosophy.

To get Django's admin app working with our models, we will perform the following steps:

1. Register our model
2. Create a super user who can access the backend
3. Run the development server
4. Access the backend in a browser

Let's register our `Movie` model with the admin by editing `django/core/admin.py`, as follows:

```
from django.contrib import admin

from core.models import Movie

admin.site.register(Movie)
```

Now our model is registered!

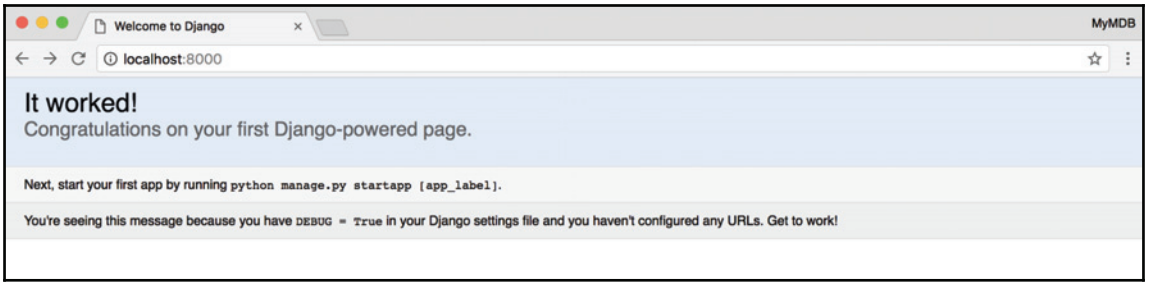
Let's now create a user who can access the backend using `manage.py`:

```
$ cd django
$ python manage.py createsuperuser
Username (leave blank to use 'tomaratyn'):
Email address: tom@aratyn.nam
Password:
Password (again):
Superuser created successfully.
```

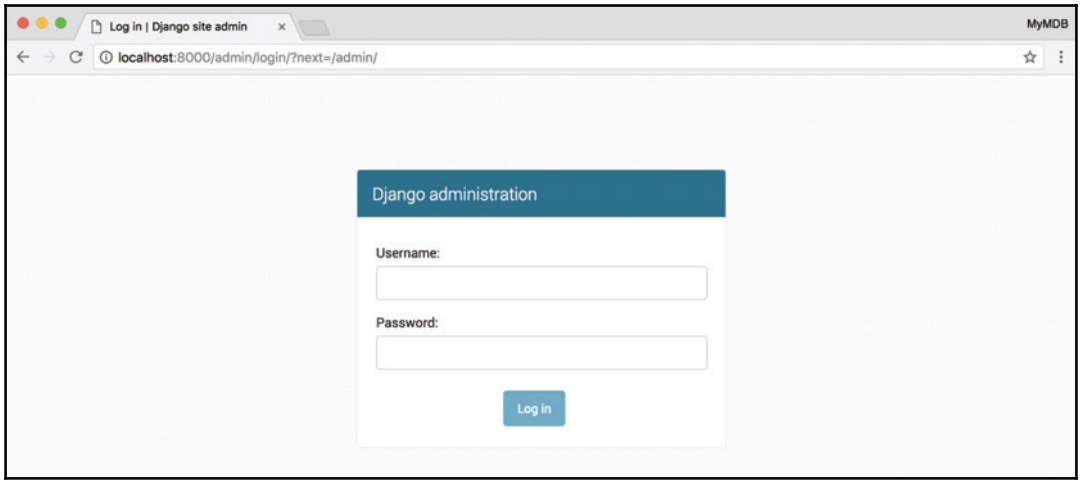
Django ships with a **development server** that can serve our app, but is not appropriate for production:

```
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
September 12, 2017 - 20:31:54
Django version 1.11.5, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

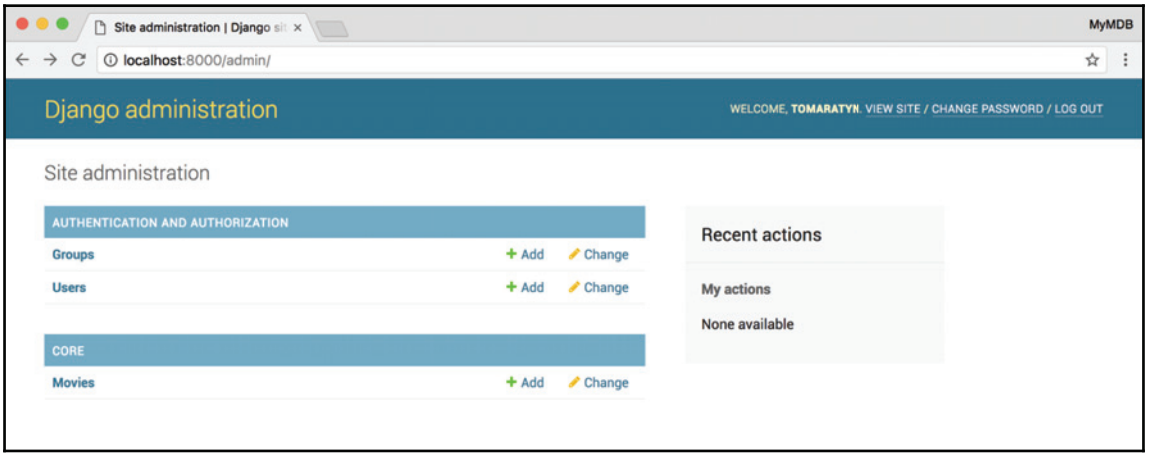
Also, open it in a browser by navigating to `http://localhost:8000/`:



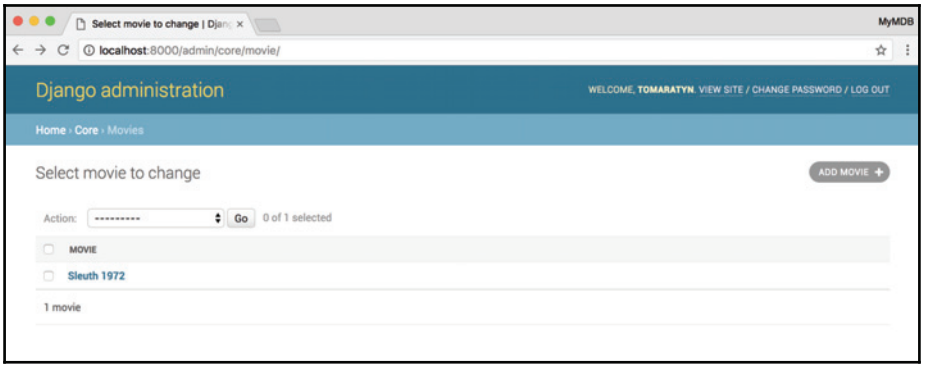
To access the admin backend, go to `http://localhost:8000/admin`:



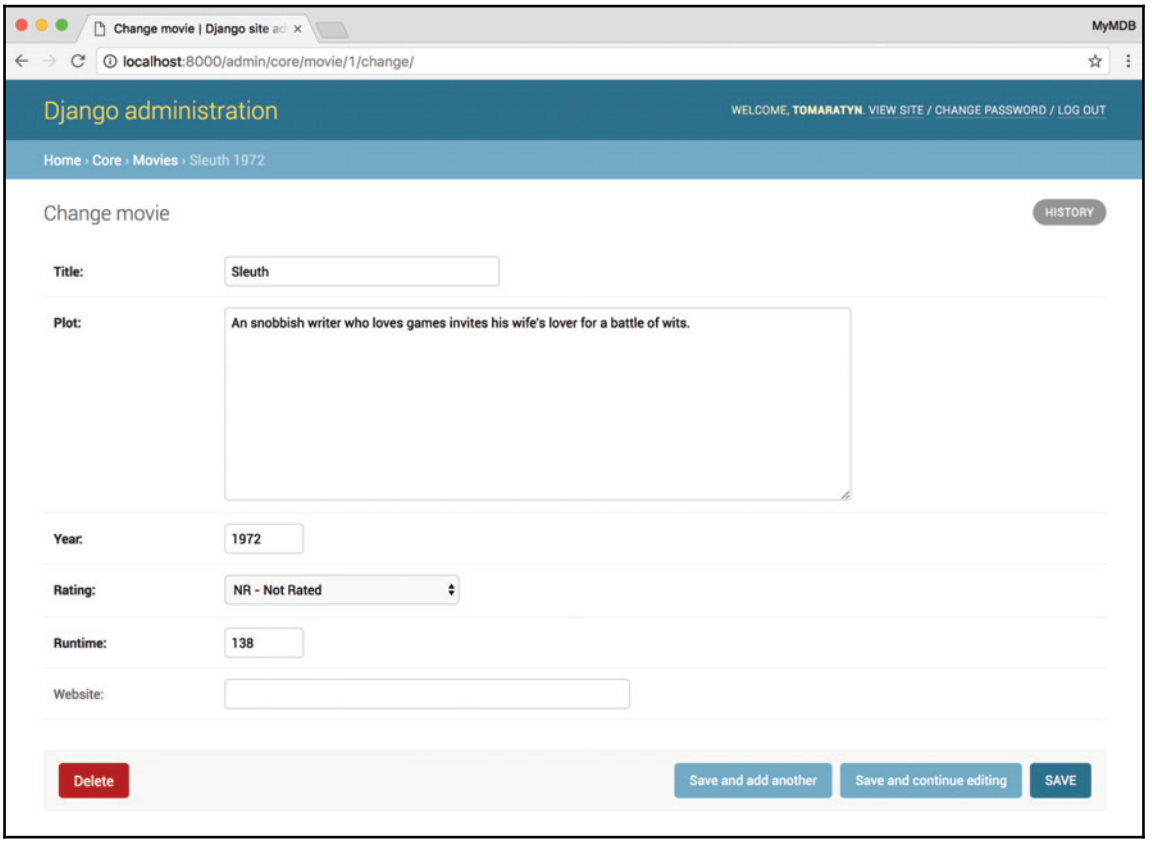
Once we log in with the credentials, we have to manage users and movies:



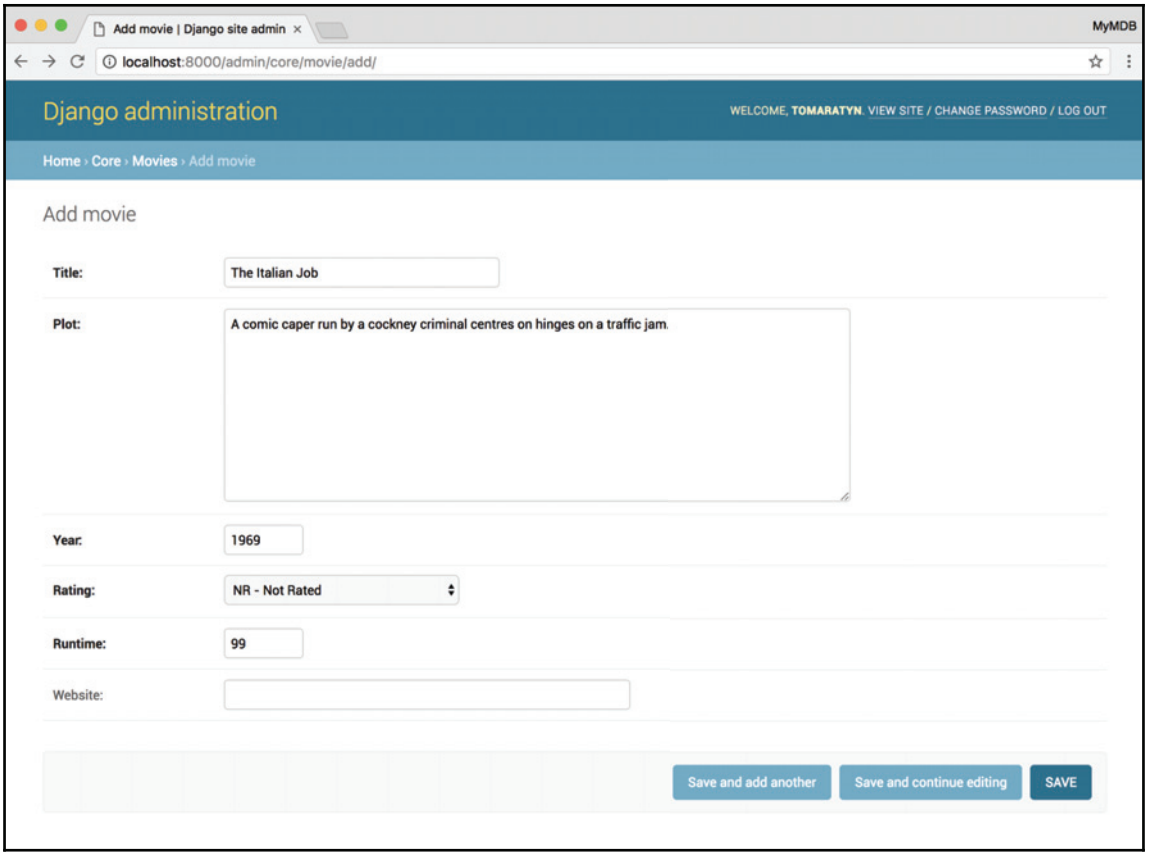
Clicking on **MOVIES** will show us a list of movies:



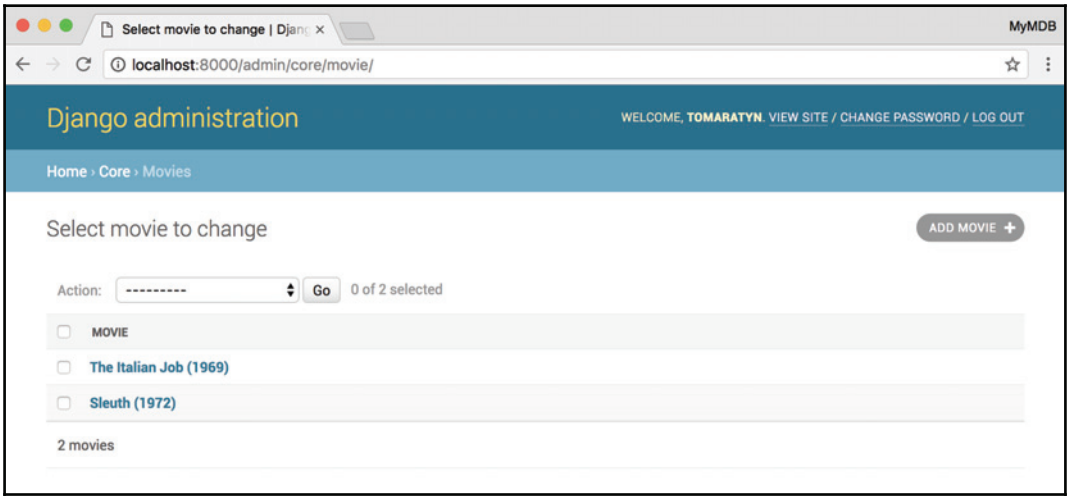
Note that the title of the link is the result of our `Movie.__str__` method. Clicking on it will give you a UI to edit the movie:



On the main admin screen and on the movie list screen, you have links to add a new movie. Let's add a new movie:



Now, our movie list shows both movies:



Now that we have a way of letting our team populate the database with movies, let's start working on the views for our users.

Creating MovieList view

When Django gets a request, it uses the path of the request and the `URLConf` of the project to match a request to a view, which returns an HTTP response. Django's views can be either functions, often referred to as **Function-Based Views** (**FBVs**), or classes, often called **Class-Based Views** (**CBVs**). The advantage of CBVs is that Django comes with a rich suite of generic views that you can subclass to easily (almost declaratively) write views to accomplish common tasks.

Let's write a view to list the movies that we have. Open `django/core/views.py` and change it to the following:

```
from django.views.generic import ListView

from core.models import Movie


class MovieList(ListView):
    model = Movie
```

`ListView` requires at least a `model` attribute. It will query for all the rows of that model, pass it to the template, and return the rendered template in a response. It also offers a number of hooks that we may use to replace default behavior, which are fully documented.

How does `ListView` know how to query all the objects in `Movie`? For that, we will need to discuss manager and `QuerySet` classes. Every model has a default manager. Manager classes are primarily used to query objects by offering methods, such as `all()`, that return a `QuerySet`. A `QuerySet` class is Django's representation of a query to the database. `QuerySet` has a number of methods, including `filter()` (such as a `WHERE` clause in a `SELECT` statement) to limit a result. One of the nice features of the `QuerySet` class is that it is lazy; it is not evaluated until we try to get a model out of the `QuerySet`. Another nice feature is that methods such as `filter()` take *lookup expressions*, which can be field names or span across relationship models. We'll be doing this throughout our projects.

All manager classes have an `all()` method that should return an unfiltered `Queryset`, the equivalent of writing `SELECT * FROM core_movie;`.

So, how does `ListView` know that it has to query all the objects in `Movie`? `ListView` checks whether it has a `model` attribute, and, if present, knows that `Model` classes have a default manager with a `all()` method, which it calls. `ListView` also gives us a convention for where to put our template, as follows: `<app_name>/<model_name>_list.html`.

Adding our first template – movie_list.html

Django ships with its own template language called the **Django Template language**. Django can also use other template languages (for example, Jinja2), but most Django projects find using the Django Template language to be efficient and convenient.

In the default configuration that is generated in our `settings.py` file, the Django Template language is configured to use `APP_DIRS`, meaning that each Django app can have a `templates` directory, which will be searched to find a template. This can be used to override templates that other apps use without having to modify the third-party apps themselves.

Let's make our first template in `django/core/templates/core/movie_list.html`:

```
<!DOCTYPE html>
<html>
  <body>
    <ul>
      {% for movie in object_list %}
        <li>{{ movie }}</li>
      {% empty %}
        <li>
          No movies yet.
        </li>
      {% endfor %}
    </ul>
    <p>
      Using https?
      {{ request.is_secure|yesno }}
    </p>
  </body>
</html>
```

Django templates are standard HTML (or whatever text format you wish to use) with variables (for example, `object_list` in our example) and tags (for example, `for` in our example). Variables will be evaluated to strings by being surrounded with `{{ }}`. Filters can be used to help format or modify variables before being printed (for example, `yesno`). We can also create custom tags and filters.

A full list of filters and tags is provided in the Django docs (`https://docs.djangoproject.com/en/2.0/ref/templates/builtins/`).

The Django template language is configured in the `TEMPLATES` variable of `settings.py`. The `DjangoTemplates` backend can take a lot of `OPTIONS`. In d*evelopment*, it can be helpful to add `'string_if_invalid': 'INVALID_VALUE',`. Any time Django can't match a variable in a template to a variable or tag, it will print out `INVALID_VALUE`, which makes it easier to catch typos. Remember that you should not use this setting in *Production*. The full list of options is available in Django's documentation (`https://docs.djangoproject.com/en/dev/topics/templates/#django.template.backends.django.DjangoTemplates`).

The final step will be to connect our view to a `URLConf`.

Routing requests to our view with URLConf

Now that we have a model, view, and template, we will need to tell Django which requests it should route to our `MovieList` View using a URLConf. Each new project has a root URLConf that created by Django (in our case it's the `django/config/urls.py` file). Django developers have developed the best practice of each app having its own URLConf. Then, the root URLConf of a project will include each app's URLConf using the `include()` function.

Let's create a URLConf for our `core` app by creating a `django/core/urls.py` file with the following code:

```
from django.urls import path

from . import views

app_name = 'core'
urlpatterns = [
    path('movies',
         views.MovieList.as_view(),
         name='MovieList'),
]
```

At its simplest, a URLConf is a module with a `urlpatterns` attribute, which is a list of `path` s. A `path` is composed of a string that describes a string, describing the path in question and a callable. CBVs are not callable, so the base `View` class has a static `as_view()` method that *returns* a callable. FBVs can just be passed in as a callback (without the `()` operator, which would execute them).

Each `path()` should be named, which is a helpful best practice for when we have to reference that path in our template. Since a URLConf can be included by another URLConf, we may not know the full path to our view. Django offers a `reverse()` function and `url` template tag to go from a name to the full path to a view.

The `app_name` variable sets the app that this `URLConf` belongs to. This way, we can reference a named `path` without Django getting confused about other apps having a `path` of the same name (for example, `index` is a very common name, so we can say `appA:index` and `appB:index` to distinguish between them).

Finally, let's connect our `URLConf` to the root `URLConf` by changing `django/config/urls.py` to the following:

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

import core.urls

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

This file looks much like our file previous `URLConf`, except that our `path()` object isn't taking a view but instead the result of the `include()` function. The `include()` function lets us prefix an entire `URLConf` with a path and give it a custom namespace.

Namespaces let us distinguish between `path` names like the `app_name` attribute does, except without modifying the app (for example, a third-party app).

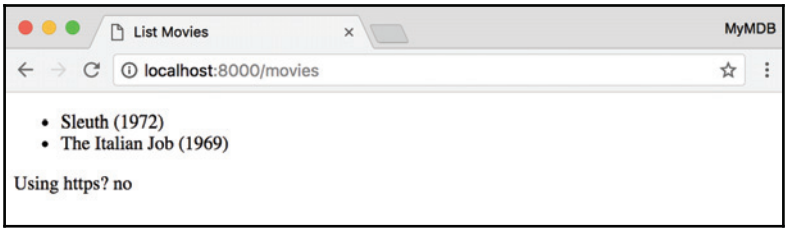
You might wonder why we're using `include()` but the Django Admin site is using `property`? Both `include()` and `admin.site.urls` return similarly formatted 3-tuple. However, instead of remembering what each portion of the 3-tuple has to have, you should just use `include()`.

Running the development server

Django now knows how to route requests to our View, which knows the Models that need to be shown and which template to render. We can tell `manage.py` to start our development server and view our result:

```
$ cd django
$ python manage.py runserver
```

In our browser, go to `http://127.0.0.1:8000/movies`:



Good job! We made our first page!

In this section, we created our first model, generated and ran the migration for it, and created a view and template so that users can browse it.

Now, let's add a page for each movie.

Individual movie pages

Now that we have our project layout, we can move more quickly. We're already tracking information for each movie. Let's create a view that will show that information.

To add movie details, we'll need to do the following things:

1. Create a `MovieDetail` view
2. Create `movie_detail.html` template
3. Reference to our `MovieDetail` view in our `URLConf`

Creating the MovieDetail view

Just like Django provides us with a `ListView` class to do all the common tasks of listing models, Django also provides a `DetailView` class that we can subclass to create a view showing the details of a single `Model`.

Let's create our view in `django/core/views.py`:

```
from django.views.generic import (
    ListView, DetailView,
)
from core.models import Movie
```

```
class MovieDetail(DetailView):
    model = Movie

class MovieList(ListView):
    model = Movie
```

A `DetailView` requires that a `path()` object include either a `pk` or `slug` in the `path` string so that `DetailView` can pass that value to the `QuerySet` to query for a specific model instance. A **slug** is a short URL-friendly label that is often used in content-heavy sites, as it is SEO friendly.

Creating the movie_detail.html template

Now that we have the View, let's make our template.

Django's Template language supports template inheritance, which means that you can write a template with all the look and feel for your website and mark the `block` sections that other templates will override. This lets us to create the look and feel of the entire website without having to edit each template. Let's use this to create a base template with MyMDB's branding and look and feel and then add a Movie Detail template that inherits from the base template.

A base template shouldn't be tied to a particular app, so let's make a general templates directory:

```
$ mkdir django/templates
```

Django doesn't know to check our `templates` directory yet, so we will need to update the configuration in our `settings.py` file. Find the line that starts with `TEMPLATES` and change the configuration to list our `templates` directory in the `DIRS` list:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'templates'),
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            # omittted for brevity
        },
    },
]
```

The only change we've made is that we added our new `templates` directory to the list under the `DIRS` key. We have avoided hardcoding the path to our `templates` directory using Python's `os.path.join()` function and the already configured `BASE_DIR`. `BASE_DIR` is set at runtime to the path of the project. We don't need to add `django/core/templates` because the `APP_DIRS` setting tells Django to check each app for the `templates` directory.

Although it's very convenient that `settings.py` is the Python file where we can use `os.path.join` and all of Python, be careful not to get too clever. `settings.py` needs to be easy to read and understand. There's nothing worse than having to debug your `settings.py`.

Let's create a base template in `django/templates/base.html` that has a main column and sidebar:

```
<!DOCTYPE html>
<html lang="en" >
<head >
  <meta charset="UTF-8" >
  <meta
    name="viewport"
    content="width=device-width, initial-scale=1, shrink-to-fit=no"
  >
  <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.mi
n.css"
    integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M"
    rel="stylesheet"
    crossorigin="anonymous"
  >
  <title >
    {% block title %}MyMDB{% endblock %}
  </title>
  <style>
    .mymdb-masthead {
      background-color: #EEEEEE;
      margin-bottom: 1em;
    }
  </style>

</head >
<body >
<div class="mymdb-masthead">
  <div class="container">
```

```
      <nav class="nav">
        <div class="navbar-brand">MyMDB</div>
        <a
          class="nav-link"
          href="{% url 'core:MovieList' %}"
        >
          Movies
        </a>
      </nav>
    </div>
  </div>

  <div class="container">
    <div class="row">
      <div class="col-sm-8 mymdb-main">
       {% block main %}{% endblock %}
      </div>
      <div
          class="col-sm-3 offset-sm-1 mymdb-sidebar"
      >
        {% block sidebar %}{% endblock %}
      </div>
    </div>
  </div>

  </body >
  </html >
```

Most of this HTML is actually bootstrap (HTML/CSS framework) boilerplate, but we do have a few new Django tags:

- `{% block title %}MyMDB{% endblock %}`: This creates a block that other templates can replace. If the block is not replaced, the contents from the parent template will be used.
- `href="{% url 'core:MovieList' %}"`: The `url` tag will produce a URL path for the named `path`. URL names should be referenced as `<app_namespace>:<name>`; in our case, `core` is the namespace of the core app (per `django/core/urls.py`), and `MovieList` is the name of the `MovieList` view's URL.

This lets us create a simple template
in `django/core/templates/core/movie_detail.html`:

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

{% block title %}
  {{ object.title }} - {{ block.super }}
{% endblock %}

{% block main %}
<h1>{{ object }}</h1>
<p class="lead">
{{ object.plot }}
</p>
{% endblock %}

{% block sidebar %}
<div>
This movie is rated:
  <span class="badge badge-primary">
  {{ object.get_rating_display }}
  </span>
</div>
{% endblock %}
```

This template has a lot less HTML in it because `base.html` already has that. All `MovieDetail.html` has to do is provide values to the blocks that `base.html` defines. Let's take a look at some new tags:

- `{% extends 'base.html' %}`: If a template wants to extend another template the first line must be an `extends` tag. Django will look for the base template (which can `extend` another template) and execute it first, then replace the blocks. A template that extends another cannot have content outside of `block`s because it's ambiguous where to put that content.
- `{{ object.title }} - {{ block.super }}`: We reference `block.super` inside the `title` template `block`. `block.super` returns the contents of the `title` template `block` in the base template.
- `{{ object.get_rating_display }}`: The Django Template language doesn't use `()` to execute the method, just referencing it by name will execute the method.

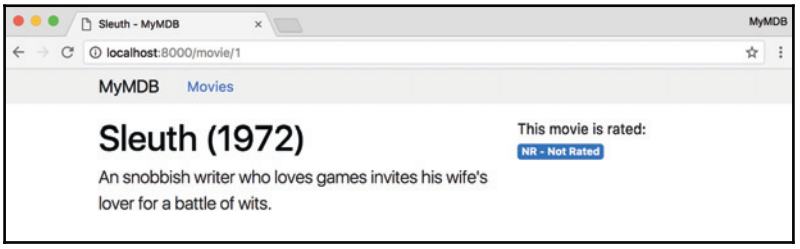
Adding MovieDetail to core.urls.py

Finally, we add our `MovieDetail` view to `core/urls.py`:

```
from django.urls import path

from . import views

urlpatterns = [
    path('movies',
         views.MovieList.as_view(),
         name='MovieList'),
    path('movie/<int:pk>',
         views.MovieDetail.as_view(),
         name='MovieDetail'),
]
```

The `MovieDetail` and `MovieList` `path()` calls both look almost the same, except for the `MovieDetail` string that has a named parameter. A `path` route string can include angle brackets to give a parameter a name (for example, `<pk>`) and even define a type that the parameter's content must conform to (for example, `<int:pk>` will only match values that parse as an `int`). These named sections are captured by Django and passed to the view by name. `DetailView` expects a `pk` (or `slug`) argument and uses it to get the correct row from the database.

Let's use `python manage.py runserver` to start the `dev` server and take a look at what our new template looks like:



A quick review of the section

In this section, we've created a new view, `MovieDetail`, learned about template inheritance, and how to pass parameters from a URL path to our view.

Next, we'll add pagination to our `MovieList` view to prevent it from querying the entire database each time.

Pagination and linking movie list to movie details

In this section, we'll update our movie list to provide a link to each movie and to have pagination to prevent our entire database being dumped into one page.

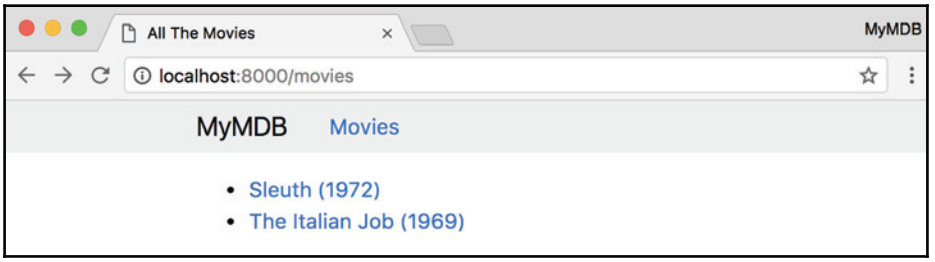
Updating MovieList.html to extend base.html

Our original `MovieList.html` was a pretty sparse affair. Let's update it to look nicer using our `base.html` template and the bootstrap CSS it provides:

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

{% block title %}
All The Movies
{% endblock %}

{% block main %}
<ul>
  {% for movie in object_list %}
    <li>
      <a href="{% url 'core:MovieDetail' pk=movie.id %}">
        {{ movie }}
      </a>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```

We're also seeing the `url` tag being used with a named argument `pk` because the `MovieDetail` URL requires a `pk` argument. If there was no argument provided, then Django would raise a `NoReverseMatch` exception on rendering, resulting in a `500` error.

Let's take a look at what it looks like:



Setting the order

Another problem with our current view is that it's not ordered. If the database is returning an unordered query, then pagination won't help navigation. What's more, there's no guarantee that each time the user changes pages that the content will be consistent, as the database may return a differently ordered result set for each time. We need our query to be ordered consistently.

Ordering our model also makes our lives as developers easier too. Whether using a debugger, writing tests, or running a shell ensuring that our models are returned in a consistent order can make troubleshooting simpler.

A Django model may optionally have an inner class called `Meta`, which lets us specify information about a Model. Let's add a `Meta` class with an `ordering` attribute:

```
class Movie(models.Model):
   # constants and fields omitted for brevity

    class Meta:
        ordering = ('-year', 'title')

    def __str__(self):
        return '{} ({})'.format(
            self.title, self.year)
```

`ordering` takes a list or tuple of, usually, strings that are field names, optionally prefixed by a – character that denotes descending order. `('-year', 'title')` is the equivalent of the SQL clause `ORDER BY year DESC, title`.

Adding `ordering` to a Model's `Meta` class will mean that `QuerySets` from the model's manager will be ordered.

Adding pagination

Now that our movies are always ordered the same way, let's add pagination. A Django `ListView` already has built-in support for pagination, so all we need to do is take advantage of it. **Pagination** is controlled by the `GET` parameter `page` that controls which page to show.

Let's add pagination to the bottom of our `main` template `block`:

```
{% block main %}
 <ul >
    {% for movie in object_list %}
      <li >
        <a href="{% url 'core:MovieDetail' pk=movie.id %}" >
          {{ movie }}
        </a >
      </li >
    {% endfor %}
  </ul >
  {% if is_paginated %}
    <nav >
      <ul class="pagination" >
        <li class="page-item" >
          <a
            href="{% url 'core:MovieList' %}?page=1"
            class="page-link"
          >
            First
          </a >
        </li >
        {% if page_obj.has_previous %}
          <li class="page-item" >
            <a
              href="{% url 'core:MovieList' %}?page={{
page_obj.previous_page_number }}"
              class="page-link"
            >
              {{ page_obj.previous_page_number }}
            </a >
          </li >
        {% endif %}
        <li class="page-item active" >
          <a
            href="{% url 'core:MovieList' %}?page={{ page_obj.number }}"
            class="page-link"
          >
            {{ page_obj.number }}
```

```
          </a >
        </li >
        {% if page_obj.has_next %}
          <li class="page-item" >
            <a
              href="{% url 'core:MovieList' %}?page={{
page_obj.next_page_number }}"
              class="page-link"
            >
              {{ page_obj.next_page_number }}
            </a >
          </li >
        {% endif %}
        <li class="page-item" >
          <a
              href="{% url 'core:MovieList' %}?page=last"
              class="page-link"
          >
            Last
          </a >
        </li >
      </ul >
    </nav >
  {% endif %}
{% endblock %}
```

Let's take a look at some important points of our `MovieList` template:

- `page_obj` is of the `Page` type, which knows information about this page of results. We use it to check whether there is a next/previous page using `has_next()`/`has_previous()` (we don't need to put `()` in the Django template language, but `has_next()` is a method, not a property). We also use it to get the `next_page_number()`/`previous_page_number()`. Note that it is important to use the `has_*()` method to check for the existence of the next/previous page numbers before retrieving them. If they don't exist when retrieved, `Page` throws an `EmptyPage` exception.
- `object_list` continues to be available and hold the correct values. Even though `page_obj` encapsulates the results for this page in `page_obj.object_list`, `ListView` does the convenient work of ensuring that we can continue to use `object_list` and our template doesn't break.

We now have the pagination working!

404 – for when things go missing

We now have a couple of views that can't function if given the wrong value in the URL (the wrong `pk` will break `MovieDetail`; the wrong `page` will break `MovieList`); let's plan for that by handling `404` errors. Django offers a hook in the root URLConf to let us use a custom view for `404` errors (also for `403`, `400`, and `500`—all following the same names scheme). In your root `urls.py` file, add a variable called `handler404` whose value is a string Python path to your custom view.

However, we can continue to use the default `404` handler view and just write a custom template. Let's add a `404` template in `django/templates/404.html`:

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

{% block title %}
Not Found
{% endblock %}

{% block main %}
<h1>Not Found</h1>
<p>Sorry that reel has gone missing.</p>
{% endblock %}
```

Even if another app throws a `404` error, this template will be used.

At the moment, if you've got an unused URL such as `http://localhost:8000/not-a-real-page`, you won't see our custom 404 template because Django's `DEBUG` settings is `True` in `settings.py`. To make our 404 template visible, we will need to change the `DEBUG` and `ALLOWED_HOSTS` settings in `settings.py`:

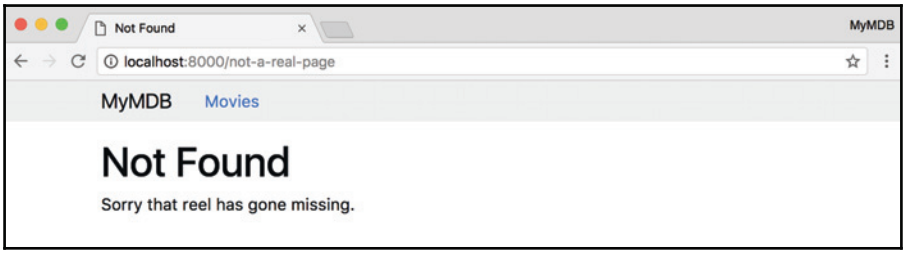
```
DEBUG = False

ALLOWED_HOSTS = [
    'localhost',
    '127.0.0.1'
]
```

`ALLOWED_HOSTS` is a setting that restricts which `HOST` values in an HTTP request Django will respond to. If `DEBUG` is `False` and a `HOST` does not match an `ALLOWED_HOSTS` value, then Django will return a `400` error (you can customize both the view and template for this error as described in the preceding code). This is a security feature that protects us and will be discussed more in our chapter on security.

Now that our project is configured, let's run the Django development server:

```
$ cd django
$ python manage.py runserver
```

With it running, we can use our web browser to open `http://localhost:8000/not-a-real-page`. Our results should look like this:



Testing our view and template

Since we now have some logic in our `MoveList` template, let's write some tests. We'll talk a lot more about testing in the `Chapter 8`, *Testing Answerly*. However, the basics are simple and follow the common XUnit pattern of the `TestCase` classes holding test methods that make assertions.

For Django's `TestRunner` to find a test, it must be in the `tests` module of an installed app. Right now, that means `tests.py`, but, eventually, you may wish to switch to a directory Python module (in which case, prefix your test filenames with `test` for the `TestRunner` to find them).

Let's add a test that performs the following functions:

- If there's more than 10 movies, then pagination controls should be rendered in the template
- If there's more than 10 movies and we don't provide `page` `GET` parameters, consider the following things:
    - The `page_is_last` context variable should be `False`
    - The `page_is_first` context variable should be `True`
    - The first item in the pagination should be marked as active

The following is our `tests.py` file:

```
from django.test import TestCase
from django.test.client import \
    RequestFactory
from django.urls.base import reverse

from core.models import Movie
from core.views import MovieList


class MovieListPaginationTestCase(TestCase):

    ACTIVE_PAGINATION_HTML = """
    <li class="page-item active">
      <a href="{}?page={}" class="page-link">{}</a>
    </li>
    """

    def setUp(self):
        for n in range(15):
            Movie.objects.create(
                title='Title {}'.format(n),
                year=1990 + n,
                runtime=100,
            )

    def testFirstPage(self):
        movie_list_path = reverse('core:MovieList')
        request = RequestFactory().get(path=movie_list_path)
        response = MovieList.as_view()(request)
        self.assertEqual(200, response.status_code)
        self.assertTrue(response.context_data['is_paginated'])
        self.assertInHTML(
            self.ACTIVE_PAGINATION_HTML.format(
                movie_list_path, 1, 1),
            response.rendered_content)
```

Let's take a look at some interesting points:

- `class MovieListPaginationTestCase(TestCase):` `TestCase` is the base class for all Django tests. It has a number of conveniences built in, including a number of convenient assert methods.

- `def setUp(self)`: Like most XUnit testing frameworks, Django's `TestCase` class offers a `setUp()` hook that is run before each test. A `tearDown()` hook is also available if needed. The database is cleaned up between each test, so we don't need to worry about deleting any models we added.
- `def testFirstPage(self)`:: A method is a test if its name is prefixed with `test`.
- `movie_list_path = reverse('core:MovieList'):reverse()` was mentioned before and is the Python equivalent of the `url` Django template tag. It will resolve the name into a path.
- `request = RequestFactory().get(path=movie_list_path):RequestFactory` is a convenient factory for creating fake HTTP requests. A `RequestFactory` has convenience methods for creating `GET`, `POST`, and `PUT` requests by its convenience methods named after the verb (for example, `get()` for `GET` requests). In our case, the `path` object provided doesn't matter, but other views may want to inspect the path of the request.
- `self.assertEqual(200, response.status_code)`: This asserts that the two arguments are equal. A response's `status_code` to check success or failure (`200` being the status code for success—the one code you never see when you browse the web).
- `self.assertTrue(response.context_data['is_paginated'])`: This asserts that the argument evaluates to `True`. `response` exposes the context that is used in rendering the template. This makes finding bugs much easier as you can quickly check actual values used in rendering.
- `self.assertInHTML(:assertInHTML` is one of the many convenient methods that Django provides as part of its **Batteries Included** philosophy. Given a valid HTML string `needle` and valid HTML string `haystack`, it will assert that `needle` is in `haystack`. The two strings need to be valid HTML because Django will parse them and examine whether one is inside the other. You don't need to worry about spacing or the order of attributes/classes. It's a very convenient assertion when you try to ensure that templates are working right.

To run tests, we can use `manage.py`:

```
$ cd django
$ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.035s

OK
Destroying test database for alias 'default'...
```

Finally, we can be confident that we've got pagination working right.

Adding Person and model relationships

In this section, we will add relationships between models to our project. People's relationship to movies can create a complex data model. The same person can be the actor, writer, and director (for example, *The Apostle* (1997) written, directed, and starring Robert Duvall). Even leaving out the crew and production teams and simplifying a bit, the data model will involve a one-to-many relationship using a `ForiengKey` field, a many-to-many relationship using a `ManyToManyField`, and a class that adds extra information about a many-to-many relationship using a `through` class in a `ManyToManyField`.

In this section, we will do the following things step by step:

1. Create a `Person` model
2. Add a `ForeignKey` field from `Movie` to `Person` to track the director
3. Add a `ManyToManyField` from `Movie` to `Person` to track the writers
4. Add a `ManyToManyField` with a `through` class (`Actor`) to track who performed and in what role in a Movie
5. Create the migration
6. Add the director, writer, and actors to the movie details template
7. Add a `PersonDetail` view to the list that indicates what movies a Person has directed, written, and performed in

Adding a model with relationships

First, we will need a `Person` class to describe and store a person involved in a movie:

```
class Person(models.Model):
    first_name = models.CharField(
        max_length=140)
    last_name = models.CharField(
        max_length=140)
    born = models.DateField()
    died = models.DateField(null=True,
                            blank=True)

    class Meta:
        ordering = (
            'last_name', 'first_name')

    def __str__(self):
        if self.died:
            return '{}, {} ({}-{})'.format(
                self.last_name,
                self.first_name,
                self.born,
                self.died)
        return '{}, {} ({})'.format(
                self.last_name,
                self.first_name,
                self.born)
```

In `Person`, we also see a new field (`DateField`) and a new parameter for fields (`null`).

`DateField` is used for tracking date-based data, using the appropriate column type on the database (`date` on Postgres) and `datetime.date` in Python. Django also offers a `DateTimeField` to store the date and time.

All fields support the `null` parameter (`False` by default), which indicates whether the column should accept `NULL` SQL values (represented by `None` in Python). We mark `died` as supporting `null` so that we can record people as living or dead. Then, in the `__str__()` method we print out a different string representation if someone is alive or dead.

We now have the `Person` model that can have various relationships with `Movies`.

Different types of relationship fields

Django's ORM has support for fields that map relationships between models, including one-to-many, many-to-many, and many-to-many with an intermediary model.

When two models have a one-to-many relationship, we use a `ForeignKey` field, which will create a column with a **Foreign Key** (**FK**) constraint (assuming that there is database support) between the two tables. In the model without the `ForeignKey` field, Django will automatically add a `RelatedManager` object as an instance attribute.
The `RelatedManager` class makes it easier to query for objects in a relationship. We'll take a look at examples of this in the following sections.

When two models have a many-to-many relationship, either (but not both) of them can get the `ManyToManyField()`; Django will create a `RelatedManager` on the other side for you. As you may know, relational databases cannot actually have a many-to-many relationship between two tables. Rather, relational databases require a *bridging* table with foreign keys to each of related tables. Assuming that we don't want to add any attributes describing the relationship, Django will create and manage this bridging table for us automatically.

Sometimes, we want extra fields to describe a many-to-many relationship (for example, when it started or ended); for that, we can provide a `ManyToManyField` with a `through` model (sometimes called an association class in UML/OO). This model will have a `ForeignKey` to each side of the relationship and any extra fields we want.

We'll create an example of each of these, as we go along adding directors, writers, and actors into our `Movie` model.

Director – ForeignKey

In our model, we will say that each movie can have one director, but each director can have directed many movies. Let's use the `ForiengKey` field to add a director to our movie:

```
class Movie(models.Model):
   # constants, methods, Meta class and other fields omitted for brevity.
    director = models.ForeignKey(
        to='Person',
        on_delete=models.SET_NULL,
        related_name='directed',
        null=True,
        blank=True)
```

Let's take a look at our new field line by line:

- `to='Person'`: All of Django's relationship fields can take a string reference as well as reference to the related model. This argument is required.
- `on_delete=models.SET_NULL`: Django needs instruction on what to do when the referenced model (instance/row) is deleted. `SET_NULL` will set the `director` field of all the `Movie` model instances directed by the deleted `Person` to `NULL`. If we wanted to cascade deletes we would use the `models.CASCADE` object.
- `related_name='directed'`: This is an optional argument that indicates the name of the `RelatedManager` instance on the other model (which lets us query all the `Movie` model instances a `Person` directed). If `related_name` were not provided, then `Person` would get an attribute called `movie_set` (following the `<model with FK>_set` pattern). In our case, we will have multiple different relationships between `Movie` and `Person` (writer, director, and actors), so `movie_set` would become ambiguous, and we must provide a `related_name`.

This is also the first time we're adding a field to an existing model. When doing so, we have to *either* add `null=True` or offer a `default` value. If we do not, then the migration will force us to. This requirement exists because Django has to assume that there are existing rows in the table (even if there aren't) when the migration is run. When a database adds the new column, it needs to know what it should insert into existing rows. In the case of the `director` field, we can accept that it may sometimes be `NULL`.

We have now added a field to `Movie` and a new attribute to `Person` instances called `directed` (of the `RelatedManager` type). `RelatedManager` is a very useful class that is like a model's default Manager, but automatically manages the relationship across the two models.

Let's take a look at `person.directed.create()` and compare it to `Movie.objects.create()`. Both methods will create a new `Movie`, but `person.directed.create()` will make sure that the new `Movie` has `person` as its `director`. `RelatedManager` also offers the `add` and `remove` methods so that we can add a `Movie` to a `directed` set of `Person` by calling `person.directed.add(movie)`. There's also a `remove()` method that works similarly, but removes a model from the relationship.

Writers – ManyToManyField

Two models may also have a many-to-many relationship, for example, a person may write many movies and a movie may be written by many people. Next, we'll add a `writers` field to our `Movie` model:

```
class Movie(models.Model):
   # constants, methods, Meta class and other fields omitted for brevity.
    writers = models.ManyToManyField(
        to='Person',
        related_name='writing_credits',
        blank=True)
```

A `ManyToManyField` established a many-to-many relationship and acts like a `RelatedManager`, permitting users to query and create models. We again use the `related_name` to avoid giving `Person` a `movie_set` attribute and instead give it a `writing_credits` attribute that will be a `RelatedManager`.

In the case of a `ManyToManyField`, both sides of the relationship have `RelatedManager` s so that `person.writing_credits.add(movie)` has the same effect as writing `movie.writers.add(person)`.

Role – ManyToManyField with a through class

The last example of a relationship field we'll look at is used when we want to use an intermediary model to describe the relationship between two other models that have a many-to-many relationship. Django lets us do this by creating a model that describes the *join table* between the two models in a many-to-many relationship.

In our case, we will create a many-to-many relationship between `Movie` and `Person` through `Role`, which will have a `name` attribute:

```
class Movie(models.Model):
   # constants, methods, Meta class and other fields omitted for brevity.
    actors = models.ManyToManyField(
        to='Person',
        through='Role',
        related_name='acting_credits',
        blank=True)

class Role(models.Model):
    movie = models.ForeignKey(Movie, on_delete=models.DO_NOTHING)
    person = models.ForeignKey(Person, on_delete=models.DO_NOTHING)
    name = models.CharField(max_length=140)
```

```
    def __str__(self):
        return "{} {} {}".format(self.movie_id, self.person_id, self.name)

    class Meta:
        unique_together = ('movie',
                           'person',
                           'name')
```

This looks like the preceding `ManyToManyField`, except we have both a `to` (referencing `Person` as before) argument and a `through` (referencing `Role`) argument.

The `Role` model looks much like one would design a *join table*; it has a `ForeignKey` to each side of the many-to-many relationship. It also has an extra field called `name` to describe the role.

`Role` also has a unique constraint on it. It requires that `movie`, `person`, and `billing` all to be unique together; setting the `unique_together` attribute on the `Meta` class of `Role` will prevent duplicate data.

This user of `ManyToManyField` will create four new `RelatedManager` instances:

- `movie.actors` will be a related manager to `Person`
- `person.acting_credits` will be a related manager to `Movie`
- `movie.role_set` will be a related manager to `Role`
- `person.role_set` will be a related manager to `Role`

We can use any of the managers to query models but only the `role_set` managers to create models or modify relationships because of the intermediary class. Django will throw an `IntegrityError` exception if you try to run `movie.actors.add(person)` because there's no way to fill in the value for `Role.name`. However, you can write `movie.role_set.add(person=person, name='Hamlet')`.

Adding the migration

Now, we can generate a migration for our new models:

```
$ python manage.py makemigrations core
Migrations for 'core':
  core/migrations/0002_auto_20170926_1650.py
    - Create model Person
    - Create model Role
    - Change Meta options on movie
```

```
    - Add field movie to role
    - Add field person to role
    - Add field actors to movie
    - Add field director to movie
    - Add field writers to movie
    - Alter unique_together for role (1 constraint(s))
```

Then, we can run our migration so that the changes get applied:

```
$ python manage.py migrate core
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0002_auto_20170926_1651... OK
```

Next, let's make our movie pages link to the people in the movies.

Creating a PersonView and updating MovieList

Let's add a `PersonDetail` view that our `movie_detail.html` template can link to. To create our view, we'll go through a four-step process:

1. Create a manager to limit the number of database queries
2. Create our view
3. Create our template
4. Create a URL that references our view

Creating a custom manager – PersonManager

Our `PersonDetail` view will list all the movies in which a `Person` is acting, writing, or directing credits. In our template, we will print out the name of each film in each credit (and `Role.name` for the acting credits). To avoid sending a flood of queries to the database, we will create new managers for our models that will return smarter `QuerySet` s.

In Django, any time we access a property across a relationship, then Django will query the database to get the related item (in the case of looping over each item `person.role_set.all()`, one for each related `Role`). In the case of a `Person` who is in *N* movies, this will result in *N* queries to the database. We can avoid this situation with the `prefetch_related()` method (later we will look at `select_related()` method). Using the `prefetch_related()` method, Django will query all the related data across a single relationship in a single additional query. However, if we don't end up using the prefetched data, querying for it will waste time and memory.

Let's create a `PersonManager` with a new method, `all_with_prefetch_movies()`, and make it the default manager for `Person`:

```
class PersonManager(models.Manager):
    def all_with_prefetch_movies(self):
        qs = self.get_queryset()
        return qs.prefetch_related(
            'directed',
            'writing_credits',
            'role_set__movie')


class Person(models.Model):
    # fields omitted for brevity

    objects = PersonManager()

    class Meta:
        ordering = (
            'last_name', 'first_name')

    def __str__(self):
        # body omitted for brevity
```

Our `PersonManager` will still offer all the same methods as the default because `PersonManager` inherits from `models.Manager`. We also define a new method, which uses `get_queryset()` to get a `QuerySet`, and tells it to prefetch the related models. `QuerySets` are lazy, so no communication with the database happens until the query set is evaluated (for example by, iteration, casting to a bool, slicing, or evaluated by an `if` statement). `DetailView` won't evaluate the query until it uses `get()` to get the model by PK.

The `prefetch_related()` method takes one or more *lookups*, and after the initial query is done, it automatically queries those related models. When you access a model related to the one from your `QuerySet`, Django won't have to query it, as you will already have it prefetched in the `QuerySet`.

A *lookup* is what a Django `QuerySet` takes to express a field or `RelatedManager` in a model. A lookup can even span across relationships by separating the name of the relationship field (or `RelatedManager`) and the related models field with two underscores:

```
Movie.objects.all().filter(actors__last_name='Freeman',
actors__first_name='Morgan')
```

The preceding call will return a `QuerySet` for all the `Movie` model instances in which Morgan Freeman has been an actor.

In our `PersonManager`, we're telling Django to prefetch all the movies that a `Person` has directed, written, and had a role in as well as prefetch the roles themselves. Using the `all_with_prefetch_movies()` method will result in a constant number of queries no matter how prolific the `Person` has been.

Creating a PersonDetail view and template

Now we can write a very thin view in `django/core/views.py`:

```
class PersonDetail(DetailView):
    queryset = Person.objects.all_with_prefetch_movies()
```

This `DetailView` is different because we're not providing it with a `model` attribute. Instead, we're giving it a `QuerySet` object from our `PersonManager` class. When `DetailView` uses the `filter()` of `QuerySet` and `get()` methods to retrieve the model instance, `DetailView` will derive the name of the template from the model instance's class name just as if we had provided model class as an attribute on the view.

Now, let's create our template in `django/core/templates/core/person_detail.html`:

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

{% block title %}
  {{ object.first_name }}
  {{ object.last_name }}
{% endblock %}

{% block main %}
```

```
    <h1>{{ object }}</h1>
    <h2>Actor</h2>
    <ul >
      {% for role in object.role_set.all %}
        <li >
          <a href="{% url 'core:MovieDetail' role.movie.id %}" >
            {{ role.movie }}
          </a >:
          {{ role.name }}
        </li >
      {% endfor %}
    </ul >
    <h2>Writer</h2>
    <ul >
      {% for movie in object.writing_credits.all %}
        <li >
          <a href="{% url 'core:MovieDetail' movie.id %}" >
            {{ movie }}
          </a >
        </li >
      {% endfor %}
    </ul >
    <h2>Director</h2>
    <ul >
      {% for movie in object.directed.all %}
        <li >
          <a href="{% url 'core:MovieDetail' movie.id %}" >
            {{ movie }}
          </a >
        </li >
      {% endfor %}
    </ul >

{% endblock %}
```

Our template doesn't have to do anything special to make use of our prefetching.

Next, we should give the `MovieDetail` view the same benefit that our `PersonDetail` view received.

Creating MovieManager

Let's start with a `MovieManager` in `django/core/models.py`:

```
class MovieManager(models.Manager):

    def all_with_related_persons(self):
        qs = self.get_queryset()
        qs = qs.select_related(
            'director')
        qs = qs.prefetch_related(
            'writers', 'actors')
        return qs


class Movie(models.Model):
    # constants and fields omitted for brevity
    objects = MovieManager()

    class Meta:
        ordering = ('-year', 'title')

    def __str__(self):
         # method body omitted for brevity
```

The `MovieManager` introduces another new method, called `select_related()`. The `select_related()` method is much like the `prefetch_related()` method but it is used when the relation leads to only one related model (for example, with a `ForeignKey` field). The `select_related()` method works by using a `JOIN` SQL query to retrieve the two models in one query. Use `prefetch_related()` when the relation *may* lead to more than one model (for example, either side of a `ManyToManyField` or a `RelatedManager` attribute).

Now, we can update our `MovieDetail` view to use the query set instead of the model directly:

```
class MovieDetail(DetailView):
    queryset = (
        Movie.objects
            .all_with_related_persons())
```

The view renders exactly the same, but it won't have to query the database each time a related `Person` model instance is required, as they were all prefetched.

A quick review of the section

In this section, we created the `Person` model and established a variety of relationships between the `Movie` and `Person` models. We created a one-to-many relationship with a `ForeignKey` field class, a many-to-many relationship using the `ManyToManyField` class, and used an intermediary (or association) class to add extra information for a many-to-many relationship by providing a `through` model to a `ManyToManyField`. We also created a `PersonDetail` view to show a `Person` model instance and used a custom model manager to control the number of queries Django sends to the database.

Summary

In this chapter, we created our Django project and started our `core` Django app. We saw how to use Django's Model-View-Template approach to create easy-to-understand code. We created concentrated database logic near the model, pagination in views, and HTML in templates following the Django best practice of *fat models, thin views,* and *dumb templates*.

Now we're ready to add users who can register and vote on their favorite movies.

2
Adding Users to MyMDB

In our preceding chapter, we started our project and created our `core` app and our `core` models (`Movie` and `Person`). In this chapter, we will build on that foundation to do the following things:

- Let users register, log in, and log out
- Let logged in users vote movies up/down
- Score each movie based on the votes
- Use votes to recommend the top 10 movies.

Let's start this chapter with managing users.

Creating the user app

In this section, you will create a new Django app, called `user`, register it with your project, and make it manage users.

At the beginning of `Chapter 1`, *Building MyMDB*, you learned that a Django project is made up of many Django apps (such as our existing `core` app). A Django app should provide well-defined and tightly scoped behavior. Adding user management to our `core` app violates that principle. Making a Django app bear too many responsibilities makes it harder to test and harder to reuse. For example, we'll be reusing the code we write in this `user` Django app throughout this book.

Creating a new Django app

As we did when we created the `core` app, we will use `manage.py` to generate our `user` app:

```
$ cd django
$ python manage.py startapp user
$ cd user
$ ls
__init__.py     admin.py        apps.py         migrations      models.py
tests.py        views.py
```

Next, we'll register it with our Django project by editing our `django/config/settings.py` file and updating the `INSTALLED_APPS` property:

```
INSTALLED_APPS = [
    'user',  # must come before admin
    'core',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

We will need to put `user` before the `admin` app for reasons that we'll discuss in the *Logging in and out* section. Generally, it's a good idea to put our apps above built-in apps.

Our `user` app is now a part of our project. Usually, we would now move on to creating and defining models for our app. However, thanks to Django's built-in `auth` app, we already have a user model that we can use.

If we want to use a custom user model, then we can register it by updating `settings.py` and setting `AUTH_USER_MODEL` to a string python path to the model (for example, `AUTH_USER_MODEL=myuserapp.models.MyUserModel`).

Next, we'll create our user registration view.

Creating a user registration view

Our `RegisterView` class will be responsible for letting users register for our site. If it receives a `GET` request, then it will show them the `UserCreationFrom`; if it gets a `POST` request, it will validate the data and create the user. `UserCreationForm` is provided by the `auth` app and provides a way to collect and validate the data required to register a user; also, it is capable of saving a new user model if the data is valid.

Let's add our view to `django/user/views.py`:

```
from django.contrib.auth.forms import (
    UserCreationForm,
)
from django.urls import (
    reverse_lazy,
)
from django.views.generic import (
    CreateView,
)


class RegisterView(CreateView):
    template_name = 'user/register.html'
    form_class = UserCreationForm
    success_url = reverse_lazy(
        'core:MovieList')
```

Let's take a look at our code line by line:

- `class RegisterView(CreateView)`:: Our view extends `CreateView`, so it doesn't have to define how to handle `GET` and `POST` requests, as we will discuss in the following steps.
- `template_name = 'user/register.html'`: This is a template that we'll create. Its context will be a little different than what we've seen before; it won't have an `object` or `object_list` variables but will have a `form` variable, which is an instance of the class we set in the `form_class` attribute.
- `form_class = UserCreationForm`: This is the form class that this `CreateView` should use. Simpler models could just say `model = MyModel`, but a user is a little more complex because passwords need to be entered twice then hashed. We'll talk about how Django stores password in `Chapter 3`, *Posters, Headshots, and Security*.

- `success_url = reverse_lazy('core:MovieList')`: When model creation succeeds, this is the URL that you need to redirect to. This is actually an optional parameter; if the model has a method called `model.get_absolute_url()`, then that will be used and we don't need to provide `success_url`.

The behavior of `CreateView` is spread across a number of base classes and mixins that interact through methods that act as hooks we can override to change behavior. Let's take a look at some of the most critical points.

If `CreateView` receives a `GET` request, it will render the template for the form. One of the ancestors of `CreateView` is `FormMixin` which overrides `get_context_data()` to call `get_form()` and add the form instance to our template's context. The rendered template is returned as the body of the response by `render_to_response`.

If `CreateView` receives a `POST` request, it will also use `get_form()` to get the form instance. The form will be *bound* to the `POST` data in the request. A bound form can validate the data it is bound to. `CreateView` will then call `form.is_valid()` and either `form_valid()` or `form_invalid()` as appropriate. `form_valid()` will call `form.save()` (saving the data to the database) then return a 302 response that will redirect the browser to `success_url`. The `form_invalid()` method will re-render the template with the form (which will now contain error messages for the user to fix and resubmit).

We're also seeing `reverse_lazy()` for the first time. It's a lazy version of `reverse()`. Lazy functions are functions that return a value that is not resolved until it is used. We can't use `reverse()` because views classes are evaluated while the full set of URLConfs are still being built, so if we need to use `reverse()` at the *class* level of a view, we must use `reverse_lazy()`. The value will not resolved until the view returns its first response.

Next, let's create the template for our view.

Creating the RegisterView template

In writing a template with a Django form, we must remember that Django doesn't provide the `<form>` or `<button type='submit>` tags, just contents of the form body. This lets us potentially include multiple Django forms in the same `<form>`. With that in mind, let's add our template to `django/user/templates/user/register.html`:

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

{% block main %}
  <h1>Register for MyMDB</h1>
```

```
    <form method="post">
      {{ form.as_p}}
      {% csrf_token %}
      <button
          type="submit"
          class="btn btn-primary">
        Register
      </button>
    </form>
  {% endblock %}
```

Like our previous templates, we extend `base.html` and put our code in one of the existing `block` s (in this case, `main`). Let's take a closer look at how forms render.

When a form is rendered, it renders in two parts, first an optional `<ul class='errorlist'>` tag of general error messages (if any), then each field is rendered in four basic parts:

- a `<label>` tag with the field name
- a `<ul class="errorlist">` tag with errors from the user's previous form submission; this will only render if there were errors for that field
- an `<input>` (or `<select>`) tag to accept input
- a `<span class="helptext">` tag for the field's help text

`Form` comes with the following three utility methods to render the form:

- `as_table()`: Each field is wrapped in a `<tr>` tag with the label in a `<th>` tag and the widget wrapped in a `<td>` tag. The containing `<table>` tag is not provided.
- `as_ul`: The entire field (label and help text widget) is wrapped in a `<li>` tag. The containing `<ul>` tag is not provided.
- `as_p`: The entire field (label and help text widget) is wrapped in a `<p>` tag.

Containing `<table>` and `<ul>` tags are not provided for the same form that a `<form>` tag is not provided, to make it easier to output multiple forms together if necessary.

If you want fine-grained control over form rendering, `Form` instances are iterable, yielding a `Field` in each iteration, or can be looked up by name as `form["fieldName"]`.

In our example, we use the `as_p()` method because we don't need fine-grained layout control.

This template is also the first time we will see the `csrf_token` tag. CSRF is a common vulnerability in web apps that we'll discuss more in `Chapter 3`, *Posters, Headshots, and Security*. Django automatically checks all `POST` and `PUT` requests for a valid `csrfmiddlewaretoken` and header. Requests missing this won't even reach the view, but will get a `403 Forbidden` response.

Now that we have our template, let's add a `path()` object to our view in our URLConf.

Adding a path to RegisterView

Our `user` app doesn't have a `urls.py` file, so we'll have to create the `django/user/urls.py` file:

```
from django.urls import path

from user import views

app_name = 'user'
urlpatterns = [
    path('register',
         views.RegisterView.as_view(),
         name='register'),
]
```

Next, we'll have to `include()` this URLConf in our root URLConf in `django/config/urls.py`:

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

import core.urls
import user.urls

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

Since URLConf will only search until the *first* matching `path` is found, we always want to put `path`s with no prefix or with the broadest URLConfs last so that they don't accidentally block other views.

Logging in and out

Django's `auth` app provides views for logging in and out. Adding this to our project will be a two-step process:

1. Registering the views in the `user` URLConf
2. Adding templates for the views

Updating user URLConf

Django's `auth` app provides a lot of views to help make user management and authentication easier, including logging in/out, changing passwords, and resetting forgotten passwords. A full-featured production app should offer all three features to users. In our case, we will restrict ourselves to just log in and log out.

Let's update `django/user/urls.py` to use log in and log out views of `auth`:

```
from django.urls import path
from django.contrib.auth import views as auth_views

from user import views

app_name = 'user'
urlpatterns = [
    path('register',
         views.RegisterView.as_view(),
         name='register'),
    path('login/',
         auth_views.LoginView.as_view(),
         name='login'),
    path('logout/',
         auth_views.LogoutView.as_view(),
         name='logout'),
]
```

If you're providing log in/log out, password change, and password reset, then you can use URLConf of `auth` as shown in the following code snippet:

```
from django.contrib.auth import urls
app_name = 'user'
urlpatterns = [
    path('', include(urls)),
]
```

Now, let's add the template.

Creating a LoginView template

First, let's add a template for the login page in `django/user/templates/registration/login.html`:

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

{% block title %}
Login - {{ block.super }}
{% endblock %}

{% block main %}
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button
        class="btn btn-primary">
      Log In
    </button>
  </form>
{% endblock %}
```

The preceding code looks very similar to `user/register.html`.

However, what should happen when the user logs in?

A successful login redirect

In `RegisterView`, we were able to specify where to redirect the user after success because we created the view. The `LoginView` class will follow these steps to decide where to redirect the user:

1. Use the `POST` parameter `next` if it is a valid URL and point at a server hosting this application. `path()` names are not available.
2. Use the `GET` parameter `next` if it is a valid URL and point at a server hosting this application. `path()` names are not available.
3. `LOGIN_REDIRECT_URL` setting which has a default of `'/accounts/profile/'`. `path()` names *are* available.

In our case, we want to redirect all users to the movie list, so let's update `django/config/settings.py` to have a `LOGIN_REDIRECT_URL` setting:

```
LOGIN_REDIRECT_URL = 'core:MovieList'
```

However, if there were cases where we wanted to redirect users to a specific page, we could use the `next` parameter to specifically redirect them to a particular page. For example, if a user tries to perform an action before they're logged in, we pass the page they were on to `LoginView` as a `next` parameter to redirect them back to the page they were on after logging in.

Now, when a user will log in, they will be redirected to our Movie List view. Next, let's create a template for the logout view.

Creating a LogoutView template

The `LogoutView` class behaves strangely. If it receives a `GET` request, it will log the user out and then try to render `registration/logged_out.html`. It's unusual for `GET` requests to modify a user's state, so it's worth remembering that this view is a bit different.

There's another wrinkle with the `LogoutView` class. If you don't provide a `registration/logged_out.html` template and you have the `admin` app installed, then Django *may* use the template of `admin` because the `admin` app does have that template (log out of the `admin` app, and you'll see it).

The way that Django resolves template names into files is a three-step process that stops as soon as a file is found, as follows:

1. Django iterates over the directories in the `DIRS` list in `settings.TEMPLATES`.
2. If `APP_DIRS` is `True`, then it will iterate over the apps listed in `INSTALLED_APPS` until a match is found. If `admin` comes before `user` in the `INSTALLED_APPS` list, then it will match first. If `user` comes first, `user` will match first.
3. Raise a `TemplateDoesNotExist` exception.

This is why we put `user` first in our list of installed apps and added a comment warning future developers not to change the order.

We're now done with our `user` app. Let's review what we've accomplished.

A quick review of the section

We've created a `user` app to encapsulate user management. In our `user` app, we leveraged a lot of functionalities that Django's `auth` app provides, including `UserCreationForm`, `LoginView`, and `LogoutView` classes. We've also learned about some new generic views that Django provides and used `CreateView` in combination with the `UserCreationForm` class to make the `RegisterView` class.

Now that we have users, let's allow them to vote on our movies.

Letting users vote on movies

Part of the fun of community sites such as IMDB is being able to vote on the movies we love and hate. In MyMDB, users will be able to vote for movies with either a 👍 or a 👎. A movie will have a score, which is the number of 👍 minus the number of 👎.

Let's start with the most important part of voting: the `Vote` model.

Creating the Vote model

In MyMDB, each user can have one vote per movie. The vote can either be positive—👍—or negative—👎.

Let's update our `django/core/models.py` file to have our `Vote` model:

```
class Vote(models.Model):
    UP = 1
    DOWN = -1
    VALUE_CHOICES = (
        (UP, "👍",),
        (DOWN, "👎",),
    )

    value = models.SmallIntegerField(
        choices=VALUE_CHOICES,
    )
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE
    )
    movie = models.ForeignKey(
        Movie,
        on_delete=models.CASCADE,
    )
    voted_on = models.DateTimeField(
        auto_now=True
    )

    class Meta:
        unique_together = ('user', 'movie')
```

This model has the following four fields:

- `value`, which must be `1` or `-1`.
- `user` is a `ForeignKey`, which references the `User` mode through `settings.AUTH_USER_MODEL`. Django recommends that you never reference `django.contrib.auth.models.User` directly but using either `settings.AUTH_USER_MODEL` or `django.contrib.auth.get_user_model()`.
- `movie` is a `ForeignKey` referencing a `Movie` model.
- `voted_on` is a `DateTimeField` with `auto_now` enabled. The `auto_now` argument makes the model update the field to the current date time every time the model is saved.

The `unique_together` attribute of Meta creates a unique constraint on the table. A unique constraint will prevent two rows having the same value for both `user` and `movie`, enforcing our rule of one vote per user per movie.

Let's create a migration for our mode with `manage.py`:

```
$ python manage.py makemigrations core
Migrations for 'core':
  core/migrations/0003_auto_20171003_1955.py
    - Create model Vote
    - Alter field rating on movie
    - Add field movie to vote
    - Add field user to vote
    - Alter unique_together for vote (1 constraint(s))
```

Then, let's run our migration:

```
$ python manage.py migrate core
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0003_auto_20171003_1955... OK
```

Now that we have our model and table set up, let's create a form to validate votes.

Creating VoteForm

Django's forms API is very robust and lets us create almost any kind of form we want. If we want to create an arbitrary form, we can create a class that extends `django.forms.Form` and add whatever fields we want to it. However, if we want to build a form that represents a model, Django offers us a shortcut with `django.forms.ModelForm`.

The type of form we want depends on where the form will be placed and how it will be used. In our case, we want a form we can place on the `MovieDetail` page and just let it give the user the following two radio buttons: 👍 and 👎.

Let's take a look at the simplest `VoteForm` possible:

```
from django import forms

from core.models import Vote

class VoteForm(forms.ModelForm):
    class Meta:
```

```
        model = Vote
        fields = (
            'value', 'user', 'movie',)
```

Django will generate a form from the `Vote` model using the `value`, `user`, and `movie` fields. `user` and `movie` will be `ModelChoiceField`s that use a `<select>` dropdown to pick the correct value, and `value` is a `ChoiceField` that also uses a `<select>` drop-down widget, not quite what we wanted by default.

`VoteForm` will require `user` and `movie`. Since we'll use `VoteForm` to save new votes, we can't eliminate those fields. However, letting users vote on behalf of other users will create a vulnerability. Let's customize our form to prevent that:

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

from core.models import Vote, Movie


class VoteForm(forms.ModelForm):

    user = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=get_user_model().
            objects.all(),
        disabled=True,
    )
    movie = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=Movie.objects.all(),
        disabled=True
    )
    value = forms.ChoiceField(
        label='Vote',
        widget=forms.RadioSelect,
        choices=Vote.VALUE_CHOICES,
    )

    class Meta:
        model = Vote
        fields = (
            'value', 'user', 'movie',)
```

In the preceding form, we've customized the fields.

Let's take a closer look at the `user` field:

- `user = forms.ModelChoiceField(`: A `ModelChoiceField` accepts another model as the value for this field. The choice of model is validated by providing a `QuerySet` instance of valid options.
- `queryset=get_user_model().objects.all(),`: A `QuerySet` that defines the valid choices for this field. In our case, any user can vote.
- `widget=forms.HiddenInput,`: The `HiddenInput` widget renders as a `<input type='hidden'>` HTML element, meaning that the user won't be distracted by any UI.
- `disabled=True,`: The `disabled` parameter tells the form to ignore any provided data for this field and only use values initially provided in the code. This prevents users from voting on behalf of other users.

The `movie` field is much the same as `user`, but with the `queryset` attribute queries for `Movie` model instances.

The value field is customized in a different way:

- `value = forms.ChoiceField(`: A `ChoiceField` is used to represent a field that can have a single value from a limited set. By default, it's represented by a drop-down list widget.
- `label='Vote',`: The `label` attribute lets us customize the label used for this field. While `value` makes sense in our code, we want users to think that their 👍/👎 is the vote.
- `widget=forms.RadioSelect,`: A dropdown hides the options until a user clicks on the dropdown. But our values are effective calls to action that we want to be always visible. Using the `RadioSelect` widget, Django will render each choice as an `<input type='radio'>` tag with the appropriate `<label>` tag and `name` value to make voting easier.
- `choices=Vote.VALUE_CHOICES,`: A `ChoiceField` must be told the valid choices; conveniently, it uses the same format as a model field's `choices` parameter, so we can reuse the `Vote.VALUE_CHOICES` tuple we used in the model.

Our newly customized form will appear with the label `vote` and two radio buttons.

Now that we have our form, let's add voting to the `MovieDetail` view and create views that know how to process votes.

Creating voting views

In this section, we will update the `MovieDetail` view to let users cast their votes and views that log the votes in the database. To process the users casting votes, we will create the following two views:

- `CreateVote`, which will be a `CreateView` to be used if a user hasn't voted for a movie yet
- `UpdateVote`, which will be an `UpdateView` to be used if a user has already voted but is changing their vote

Let's start by updating `MovieDetail` to provide a UI for voting on a movie.

Adding VoteForm to MovieDetail

Our `MovieDetail.get_context_data` method will be a bit more complex now. It will have to get the user's vote for the movie, instantiate the form, and know which URL to submit the vote to (`create_vote` or `update_vote`).

The first thing we will need is a way to check whether a user model has a related `Vote` model instance for a given `Movie` model instance. To do this, we will create a `VoteManager` class with a custom method. Our method will have a special behavior—if there is no matching `Vote` model instance, it will return an *unsaved* blank `Vote` object. This will make it easier to instantiate our `VoteForm` with the proper `movie` and `user` values.

Here's our new `VoteManager`:

```
class VoteManager(models.Manager):

    def get_vote_or_unsaved_blank_vote(self, movie, user):
        try:
            return Vote.objects.get(
                movie=movie,
                user=user)
        except Vote.DoesNotExist:
            return Vote(
                movie=movie,
                user=user)
```

```
class Vote(models.Model):
    # constants and field omitted

    objects = VoteManager()

    class Meta:
        unique_together = ('user', 'movie')
```

`VoteManager` is much like our previous `Managers`.

One thing we haven't encountered before is instantiating a model using its constructor (for example, `Vote(movie=movie, user=user)`) as opposed to its manager's `create()` method. Using the constructor creates a new model in memory but *not* in the database. An unsaved model is fully functional in itself (all the methods and manager methods are generally available), with the exception of anything that relies on relationships. An unsaved model has no `id` thus cannot be looked up using a `RelatedManager` or `QuerySet` until it is saved by calling its `save()` method.

Now that we have everything that `MovieDetail` needs, let's update it:

```
class MovieDetail(DetailView):
    queryset = (
        Movie.objects
           .all_with_related_persons())

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        if self.request.user.is_authenticated:
            vote = Vote.objects.get_vote_or_unsaved_blank_vote(
                movie=self.object,
                user=self.request.user
            )
                    if vote.id:
                vote_form_url = reverse(
                    'core:UpdateVote',
                    kwargs={
                        'movie_id': vote.movie.id,
                        'pk': vote.id})
            else:
                vote_form_url = (
                    reverse(
                        'core:CreateVote',
                        kwargs={
                            'movie_id': self.object.id}
                    )
                )
```

```
            vote_form = VoteForm(instance=vote)
            ctx['vote_form'] = vote_form
            ctx['vote_form_url'] = \
                vote_form_url
        return ctx
```

We've introduced two new elements in the preceding code, `self.request` and instantiating forms with instances.

Views have access to the request that they're processing through their `request` attribute. Also, `Request`s have a `user` property that gives us access to the user who made the request. We use this to check whether the user is authenticated or not, since only authenticated users can vote.

`ModelForms` can be instantiated with an instance of the model they represent. When we instantiate a `ModelForm` with an instance and render it, the fields will have the values of the instance. A nice shortcut for a common task is to display this model's values in this form.

We will also reference two `path`s that we haven't created yet; we'll do that in a moment. First, let's finish off our `MovieDetail` update by updating the `movie_detail.html` template sidebar block:

```
{% block sidebar %}
 {# rating div omitted #}
  <div>
    {% if vote_form %}
      <form
          method="post"
          action="{{ vote_form_url }}" >
        {% csrf_token %}
        {{ vote_form.as_p }}
        <button
            class="btn btn-primary" >
          Vote
        </button >
      </form >
    {% else %}
      <p >Log in to vote for this
        movie</p >
    {% endif %}
  </div >
{% endblock %}
```

In designing this, we again follow the principle that templates should have the least amount of logic possible.

Next, let's add our `CreateVote` view.

Creating the CreateVote view

The `CreateVote` view will be responsible for validating vote data using `VoteForm` and then creating the correct `Vote` model instance. However, we will not create a template for voting. If there's a problem, we'll just redirect the user to the `MovieDetail` view.

Here's the `CreateVote` view we should have in our `django/core/views.py` file:

```
from django.contrib.auth.mixins import (
    LoginRequiredMixin, )
from django.shortcuts import redirect
from django.urls import reverse
from django.views.generic import (
    CreateView, )

from core.forms import VoteForm


class CreateVote(LoginRequiredMixin, CreateView):
    form_class = VoteForm

    def get_initial(self):
        initial = super().get_initial()
        initial['user'] = self.request.user.id
        initial['movie'] = self.kwargs[
            'movie_id']
        return initial

    def get_success_url(self):
        movie_id = self.object.movie.id
        return reverse(
            'core:MovieDetail',
            kwargs={
                'pk': movie_id})

    def render_to_response(self, context, **response_kwargs):
        movie_id = context['object'].id
        movie_detail_url = reverse(
            'core:MovieDetail',
            kwargs={'pk': movie_id})
        return redirect(
            to=movie_detail_url)
```

We've introduced four new concepts in the preceding code that are different than in the `RegisterView` class—`get_initial()`, `render_to_response()`, `redirect()`, and `LoginRequiredMixin`. They are as follows:

- `get_initial()` is used to pre-populate a form with `initial` values before the form gets `data` values from the request. This is important for `VoteForm` because we've disabled `movie` and `user`. `Form` disregards `data` assigned to disabled fields. Even if a user sends in a different `movie` value or `user` value in the form, it will be disregarded by the disabled fields, and our `initial` values will be used instead.
- `render_to_response()` is called by `CreateView` to return a response with the render template to the client. In our case, we will not return a response with a template, but an HTTP redirect to `MovieDetail`. There is a serious downside to this approach—we lose any errors associated with the form. However, since our user has only two choices for input, there aren't many error messages we could provide anyway.
- `redirect()` is from Django's `django.shortcuts` package. It provides shortcuts for common operations, including creating an HTTP redirect response to a given URL.
- `LoginRequiredMixin` is a mixin that can be added to any `View` and will check whether the request is being made by an authenticated user. If the user is not logged in, they will be redirected to the login page.

Django's default setting for a login page is `/accounts/profile/`, so let's change this by editing our `settings.py` file and adding a new setting:

```
LOGIN_REDIRECT_URL = 'user:login'
```

We now have a view that will create a `Vote` model instance and redirect the user back to the related `MovieDetail` view on success or failure.

Next, let's add a view to let users update their `Vote` model instances.

Creating the UpdateVote view

The `UpdateVote` view is much simpler because `UpdateView` (like `DetailView`) takes care of the job of looking up the vote though we still have to be concerned about `Vote` tampering.

Let's update our `django/core/views.py` file:

```
from django.contrib.auth.mixins import (
    LoginRequiredMixin, )
from django.core.exceptions import (
    PermissionDenied)
from django.shortcuts import redirect
from django.urls import reverse
from django.views.generic import (
    UpdateView, )

from core.forms import VoteForm

class UpdateVote(LoginRequiredMixin, UpdateView):
    form_class = VoteForm
    queryset = Vote.objects.all()

    def get_object(self, queryset=None):
        vote = super().get_object(
            queryset)
        user = self.request.user
        if vote.user != user:
            raise PermissionDenied(
                'cannot change another '
                'users vote')
        return vote

    def get_success_url(self):
        movie_id = self.object.movie.id
        return reverse(
            'core:MovieDetail',
            kwargs={'pk': movie_id})

    def render_to_response(self, context, **response_kwargs):
        movie_id = context['object'].id
        movie_detail_url = reverse(
            'core:MovieDetail',
            kwargs={'pk': movie_id})
        return redirect(
            to=movie_detail_url)
```

Our `UpdateVote` view checks whether the `Vote` retrieved is the logged in user's vote in the `get_object()` method. We've added this check to prevent vote tampering. Our user interface doesn't let users do this by mistake. If the `Vote` wasn't cast by the logged in user then `UpdateVote` throws a `PermissionDenied` exception that Django will process and return into a `403 Forbidden` response.

The final step will be to register our new views with the `core` URLConf.

Adding views to core/urls.py

We've now created two new views, but, as always, they're not accessible to users until they're listed in a URLConf. Let's edit `core/urls.py`:

```
urlpatterns = [
    # previous paths omitted
    path('movie/<int:movie_id>/vote',
         views.CreateVote.as_view(),
         name='CreateVote'),
    path('movie/<int:movie_id>/vote/<int:pk>',
         views.UpdateVote.as_view(),
         name='UpdateVote'),
]
```

A quick review of the section

In this section, we saw examples of how to build basic and highly customized forms for accepting and validating user input. We also discussed some of the built-in views that simplify the common tasks of processing forms.

Next, we'll show how to start using our users, votes to rank each movie and provide a top-10 list.

Calculating Movie score

In this section, we'll use Django's aggregate query API to calculate the score for each movie. Django makes writing database agnostic aggregate queries easy by building the functionality into its `QuerySet` objects.

Let's start by adding a method to calculate a score to `MovieManager`.

Using MovieManager to calculate Movie score

Our `MovieManager` class is responsible for building `QuerySet` objects associated with `Movie`. We now need a new method that retrieves movies (ideally still with the related persons) and marking each movie's score based on the sum of the votes it received (we can just sum all the `1` and `-1`).

Let's take a look at how we can do this using Django's `QuerySet.annotate()` API:

```
from django.db.models.aggregates import (
    Sum
)


class MovieManager(models.Manager):

    def all_with_related_persons(self):
        qs = self.get_queryset()
        qs = qs.select_related(
            'director')
        qs = qs.prefetch_related(
            'writers', 'actors')
        return qs

    def all_with_related_persons_and_score(self):
        qs = self.all_with_related_persons()
        qs = qs.annotate(score=Sum('vote__value'))
        return qs
```

In `all_with_related_persons_and_score`, we call `all_with_related_persons` and get a `QuerySet` that we can modify further with our `annotate()` call.

`annotate` turns our regular SQL query into an aggregate query, adding the supplied aggregate operation's result to a new attribute called `score`. Django abstracts most common SQL aggregate functions into class representations, including `Sum`, `Count`, and `Average` (and many more).

The new `score` attribute is available on any instance we `get()` out of the `QuerySet` as well as in any methods we want to call on our new `QuerySet` (for example, `qs.filter(score__gt=5)` would return a `QuerySet` that has movies with a `score` attribute greater than 5).

Our new method still returns a `QuerySet` that is lazy, which means that our next step is to update `MovieDetail` and its template.

Updating MovieDetail and template

Now that we can query movies with their scores, let's change the `QuerySet` `MovieDetail` uses:

```
class MovieDetail(DetailView):
   queryset = Movie.objects.all_with_related_persons_and_score()
   def get_context_data(self, **kwargs):
       # body omitted for brevity
```

Now, when `MovieDetail` uses `get()` on its query set, the `Movie` will have a score attribute. Let's use it in our `movie_detail.html` template:

```
{% block sidebar %}
  {# movie rating div omitted #}
  <div >
    <h2 >
      Score: {{ object.score|default_if_none:"TBD" }}
    </h2 >
  </div>
  {# voting form div omitted #}
{% endblock %}
```

We can reference the `score` property safely because of `QuerySet` of `MovieDetail`. However, we don't have a guarantee that the score will not be `None` (for example, if the `Movie` has no votes). To guard against a blank score, we use the `default_if_none` filter to provide a value to print out.

We now have a `MovieManager` method that can calculate the score for all movies, but when you use it in `MovieDetail`, it means that it will only do so for the `Movie` being displayed.

Summary

In this chapter, we added users to our system, letting them register, log in (and out), and vote on our movies. We learned how to use aggregate queries to efficiently calculate the results of these votes in the database.

Next, we will let users upload pictures associated with our `Movie` and `People` models and discuss security considerations.

3
Posters, Headshots, and Security

Movies are a visual medium, so a database of movies should, at the very least, have images. Letting users upload files can have big security implications; so, in this chapter, we'll discuss both topics together.

In this chapter, we will do the following things:

- Add a file upload functionality that lets users upload images for each movie
- Examine the **Open Web Application Security Project** (**OWASP**) top 10 list of risks

We'll examine the security implications of the file upload as we go. Also, we'll take a look at where Django can help us and where we have to make careful design decisions.

Let's start by adding file upload to MyMDB.

Uploading files to our app

In this section, we will create a model that will represent and manage the files that our users upload to our site; then, we'll build a form and view to validate and process those uploads.

Configuring file upload settings

Before we begin implementing file upload, we will need to understand that file upload depends on a number of settings that must be different in production and development. These settings affect how files are stored and served.

Django has two sets of settings for files: `STATIC_*` and `MEDIA_*`. **Static files** are files that are part of our project, developed by us (for example, CSS and JavaScript). **Media files** are files that users upload to our system. Media files should not be trusted and certainly *never* executed.

We will need to set two new settings in our `django/conf/settings.py`:

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

`MEDIA_URL` is the URL that will serve the uploaded files. In development, the value doesn't matter very much, as long as it doesn't conflict with the URL of one of our views. In production, uploaded files should be served from a different domain (not a subdomain) than the one that serves our app. A user's browser that gets tricked into executing a file it requested from the same domain (or a subdomain) as our app will trust that file with the cookies (including the session ID) for our user. This default policy of all browsers is called the **Same Origin Policy**. We'll discuss this again in `Chapter 5`, *Deploying with Docker*.

`MEDIA_ROOT` is the path to the directory where Django should save the code. We want to make sure that this directory is not under our code directory so that it won't be accidentally checked in to version control or accidentally granted any generous permissions (for example, execution permission) that we grant our code base.

There are other settings we will want to configure in production, such as limiting the request body, but those will be done as part of deployment in `Chapter 5`, *Deploying with Docker*.

Next, let's create that `media_root` directory:

```
$ mkdir media_root
$ ls
django                  media_root              requirements.dev.txt
```

Great! Next, let's create our `MovieImage` model.

Creating the MovieImage model

Our `MovieImage` model will use a new field called `ImageField` to save the file and to *attempt* to validate that a file is an image. Although `ImageField` does try to validate the field, it is not enough to stop a malicious user who crafts an intentionally malicious file (but will help a user who accidentally clicked on a `.zip` instead of a `.png`). Django uses the `Pillow` library to do this validation; so, let's add `Pillow` to our requirements file `requirements.dev.txt`:

```
Pillow<4.4.0
```

Then, install our dependencies with `pip`:

```
$ pip install -r requirements.dev.txt
```

Now, we can create our model:

```
from uuid import uuid4

from django.conf import settings
from django.db import models


def movie_directory_path_with_uuid(
        instance, filename):
    return '{}/{}'.format(
        instance.movie_id, uuid4())


class MovieImage(models.Model):
    image = models.ImageField(
        upload_to=movie_directory_path_with_uuid)
    uploaded = models.DateTimeField(
        auto_now_add=True)
    movie = models.ForeignKey(
        'Movie', on_delete=models.CASCADE)
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE)
```

`ImageField` is a specialized version of `FileField` that uses `Pillow` to confirm that a file is an image. `ImageField` and `FileField` work with Django's file storage API, which provides a way to store and retrieve files, as well as read and write them. By default, Django ships with `FileSystemStorage`, which implements the storage API to store data on the local filesystem. This is sufficient for development, but we'll look at alternatives in `Chapter 5`, *Deploying with Docker*.

We used the `upload_to` parameter of `ImageField` to specify a function to generate the uploaded file's name. We don't want users to be able to specify the name of files in our system, as they may choose names that abuse our users' trust and make us look bad. We use a function that will store all the images for a given movie in the same directory and use `uuid4` to generate a universally unique name for each file (this also avoids name collisions and dealing with files overwriting each other).

We also record who uploaded the file so that if we find a bad file, we have a clue for how to find other bad files.

Let's now make a migration and apply it:

```
$ python manage.py makemigrations core
Migrations for 'core':
  core/migrations/0004_movieimage.py
    - Create model MovieImage
$ python manage.py migrate core
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0004_movieimage... OK
```

Next, let's build a form for our `MovieImage` model and use it in our `MovieDetail` view.

Creating and using the MovieImageForm

Our form will be much like our `VoteForm` in that it will hide and disable the `movie` and `user` fields that are necessary for our model but dangerous to trust from the client. Let's add it to `django/core/forms.py`:

```
from django import forms

from core.models import MovieImage


class MovieImageForm(forms.ModelForm):
```

```
    movie = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=Movie.objects.all(),
        disabled=True
    )

    user = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=get_user_model().
            objects.all(),
        disabled=True,
    )

    class Meta:
        model = MovieImage
        fields = ('image', 'user', 'movie')
```

We don't override the `image` field with a custom field or widget because the `ModelForm` class will automatically provide the correct `<input type="file">`.

Now, we can use it in the `MovieDetail` view:

```
from django.views.generic import DetailView

from core.forms import (VoteForm,
    MovieImageForm,)
from core.models import Movie


class MovieDetail(DetailView):
    queryset = Movie.objects.all_with_related_persons_and_score()

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        ctx['image_form'] = self.movie_image_form()
        if self.request.user.is_authenticated:
            # omitting VoteForm code.
        return ctx

 def movie_image_form(self):
        if self.request.user.is_authenticated:
            return MovieImageForm()
        return None
```

This time, our code is simpler because users can *only* upload new images, no other operations are supported, letting us always provide an empty form. However, with this approach, we still don't show error messages. Losing error messages should not be viewed as best practice.

Next, we'll update our template to use our new form and uploaded images.

Updating movie_detail.html to show and upload images

We'll have to make two updates to our `movie_detail.html` template. First, we will need to update our `main` template `block` to have a list of images. Second, we'll have to update our `sidebar` template `block` to contain our upload form.

Let's update our `main` block first:

```
{% block main %}
  <div class="col" >
    <h1 >{{ object }}</h1 >
    <p class="lead" >
      {{ object.plot }}
    </p >
  </div >
  <ul class="movie-image list-inline" >
    {% for i in object.movieimage_set.all %}
      <li class="list-inline-item" >
          <img src="{{ i.image.url }}" >
      </li >
    {% endfor %}
  </ul >
  <p >Directed
    by {{ object.director }}</p >
 {# writers and actors html omitted #}
{% end block %}
```

We used the `image` field's `url` property in the preceding code, which returns the `MEDIA_URL` setting joined with the calculated filename so that our `img` tag correctly displays the image.

In the `sidebar block`, we'll add our form to upload a new image:

```
{% block sidebar %}
  {# rating div omitted #}
  {% if image_form %}
    <div >
      <h2 >Upload New Image</h2 >
      <form method="post"
            enctype="multipart/form-data"
            action="{% url 'core:MovieImageUpload' movie_id=object.id %}" >
        {% csrf_token %}
        {{ image_form.as_p }}
        <p >
          <button
              class="btn btn-primary" >
            Upload
          </button >
        </p >
      </form >
    </div >
  {% endif %}
  {# score and voting divs omitted #}
{% endblock %}
```

This is very similar to our preceding form. However, we *must* remember to include the `enctype` property in our `form` tag for the uploaded file to be attached to the request properly.

Now that we're done with our template, we can create our `MovieImageUpload` view to save our uploaded files.

Writing the MovieImageUpload view

Our penultimate step will be to add a view to process the uploaded file to `django/core/views.py`:

```
from django.contrib.auth.mixins import (
    LoginRequiredMixin)
from django.views.generic import CreateView

from core.forms import MovieImageForm


class MovieImageUpload(LoginRequiredMixin, CreateView):
    form_class = MovieImageForm
```

```
    def get_initial(self):
        initial = super().get_initial()
        initial['user'] = self.request.user.id
        initial['movie'] = self.kwargs['movie_id']
        return initial

    def render_to_response(self, context, **response_kwargs):
        movie_id = self.kwargs['movie_id']
        movie_detail_url = reverse(
            'core:MovieDetail',
            kwargs={'pk': movie_id})
        return redirect(
            to=movie_detail_url)

    def get_success_url(self):
        movie_id = self.kwargs['movie_id']
        movie_detail_url = reverse(
            'core:MovieDetail',
            kwargs={'pk': movie_id})
        return movie_detail_url
```

Our view once again delegates all the work of validating and saving the model to `CreateView` and our form. We retrieve the `user.id` attribute from the request's `user` property (certain that the user is logged in because of the `LoginRequiredMixin` class) and the movie ID from the URL, then pass them to the form as initial arguments since the `user` and `movie` fields of `MovieImageForm` are disabled (so they ignore the values from the request body). The work of saving and renaming the file is all done by Django's `ImageField`.

Finally, we can update our project to route requests to our `MovieImageUpload` view and serve our uploaded files.

Routing requests to views and files

In this section, we'll update `URLConf` of `core` to route requests to our new `MovieImageUpload` view and look at how we can serve our uploaded images in development. We'll take a look at how to serve the uploaded images in production `Chapter 5`, *Deploying with Docker*.

To route requests to our `MovieImageUpload` view, we'll update `django/core/urls.py`:

```
from django.urls import path

from . import views

app_name = 'core'
urlpatterns = [
    # omitted existing paths
    path('movie/<int:movie_id>/image/upload',
         views.MovieImageUpload.as_view(),
         name='MovieImageUpload'),
    # omitted existing paths
]
```

We add our `path()` function as usual, and ensure that we remember that it expects a parameter called `movie_id`.

Now, Django will know how to route to our view, but it doesn't know how to serve the uploaded files.

To serve the uploaded files in development, we'll update `django/config/urls.py`:

```
from django.conf import settings
from django.conf.urls.static import (
    static, )
from django.contrib import admin
from django.urls import path, include

import core.urls
import user.urls

MEDIA_FILE_PATHS = static(
    settings.MEDIA_URL,
    document_root=settings.MEDIA_ROOT)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('user/', include(
        user.urls, namespace='user')),
    path('', include(
        core.urls, namespace='core')),
] + MEDIA_FILE_PATHS
```

Django offers the `static()` function, which will return a list with a single `path` object that will route any request beginning with the string `MEDIA_URL` to a file inside `document_root`. It will give us a way of serving our uploaded image files in development. This feature is not appropriate for production, and `static()` will return an empty list if `settings.DEBUG` is `False`.

Now that we've seen much of Django core functionality, let's discuss how it relates to the **Open Web Application Security Project** (**OWASP**) list of the top 10 most critical security risks (OWASP Top 10).

OWASP Top 10

The OWASP is a not for profit charitable organization focused on making *security visible* by providing impartial practical security advice for web applications. All of OWASP's materials are free and open source. Since 2010, OWASP solicits data from information security professionals and uses it to develop a list of the top 10 most critical security risks in web application security (the OWASP Top 10). Although this list does not claim to enumerate all problems (it's just the top 10), it is based on what security professionals are seeing out in the wild while doing penetration tests and code audits on real code either in production or development at companies around the world.

Django is developed to minimize and avoid these risks as much as possible and, where possible, to give developers the tools to minimize the risks themselves.

Let's enumerate the OWASP Top 10 from 2013 (the latest version at the time of writing, the 2017 RC1 having been rejected) and take a look at how Django helps us mitigate each risk.

A1 injection

This has been the number one issue since the creation of the OWASP Top 10. **Injection** means users being able to inject code that is executed by our system or a system we use. For example, SQL Injection vulnerabilities let an exploiter execute arbitrary SQL code in our database, which can lead to them circumventing almost all the controls and security measures we have (for example, letting them authenticate as an administrative user; SQL Injection exploits may lead to shell access). The best solution for this, particularly for SQL Injection, is to use parametrized queries.

Django protects us from SQL Injection by providing us with the `QuerySet` class. `QuerySet` ensures that all queries it sends are parameterized so that the database is able to distinguish between our SQL code and the values in the queries. Using parametrized queries will prevent SQL Injection.

However, Django does permit raw SQL queries using `QuerySet.raw()` and `QuerySet.extra()`. Both these methods support parameterized queries, but it is up to the developer to ensure that they **never** put values from a user into a SQL query using string formatting (for example, `str.format`) but **always** use parameters.

A2 Broken Authentication and Session Management

Broken Authentication and **Session Management** refer to the risk of an attacker being able to either authenticate as another user or take over another user's session.

Django protects us here in a few ways, as follows:

- Django's `auth` app always hashes and salts passwords so that even if the database is compromised, user passwords cannot be reasonably cracked.
- Django supports multiple *slow* hashing algorithms (for example, Argon2 and Bcrypt), which make brute-force attacks impractical. These algorithms are not provided out of the box (Django uses `PBDKDF2` by default) because they rely on third-party libraries but can be configured using the `PASSWORD_HASHERS` setting.
- The Django session ID is never made available in the URL by default, and the session ID changes after login.

However, Django's cryptographic functionality is always seeded with the `settings.SECRET_KEY` string. Checking production value of `SECRET_KEY` into version control should be considered a security problem. The value should never be shared in plain text, as we'll discuss `Chapter 5`, *Deploying with Docker*.

A3 Cross Site Scripting

Cross Site Scripting (**XSS**) is when an attacker is able to get a web app to display HTML or JavaScript created by the attacker rather than the one created by the developer(s). This attack is very powerful because if the attacker can execute arbitrary JavaScript, then they can send requests, which look indistinguishable from genuine requests from the user.

Django protects all variables in templates with HTML encoding by default.

However, Django does provide utilities to mark text as safe, which will result in values not being encoded. These should be used sparingly and with a full appreciation for the dire security consequences if they are abused.

A4 insecure direct object references

Insecure direct object references are when we insecurely expose implementation details in our resource references without protecting the resources from illicit access/exploitation. For example, the paths in the `src` attribute of our movie detail page's `<img>` tag map directly to files in the filesystem. If a user manipulates a URL, they could access images to which they should not have access to, thus exploiting a vulnerability. Or, using auto incrementing primary keys that are exposed to the user in a URL can let malicious users iterate through all the items in the database. The impact of this risk is highly dependent on the resources exposed.

Django helps us by not coupling routing paths to views. We can do model lookups based on primary keys, but we are not required to do so and may add extra fields to our models (for example, `UUIDField`) to decouple table primary keys from IDs used in URLs. In our Mail Ape project in Part 3, we'll see how we can use the `UUIDField` class as the primary key of a model.

A5 Security misconfiguration

Security misconfiguration refers to the risk incurred when the proper security mechanisms are deployed inappropriately. This risk is at the border of development and operations and requires the two teams to cooperate. For example, if we run our Django app in production with the `DEBUG` setting set to `True`, we would risk exposing far too much information to the public without having any errors in our code base.

Django helps us with sane defaults and technical and topic guides on the Django project website. The Django community is also helpful—they post on mailing lists and online blogs, though online blog posts should be treated skeptically until you validate their claims.

A6 Sensitive data exposure

Sensitive data exposure is the risk that sensitive data may be accessed without the proper authorization. This risk is broader than just an attacker highjacking a user's session, as it includes questions of how backups are stored, how encryption keys are rotated, and, most importantly, which data is actually considered *sensitive*. The answers to these questions are project/business specific.

Django can help reduce risks of inadvertent exposure from attackers using network sniffing by being configured to serve pages only over HTTPS.

However, Django doesn't provide encryption directly nor does it manage key rotation, logs, backups, and the database itself. There are many factors that affect this risk, which are outside of Django's scope.

A7 Missing function level access control

While A6 referred to data being exposed, missing function level access control refers to functionality being inadequately protected. Consider our `UpdateVote` view—if we had forgotten the `LoginRequiredMixin` class, then anyone could send an HTTP request and change our users' votes.

Django's `auth` app provides a lot of useful features to mitigate these issues, including a permission system that is outside the scope of this project and mixins and utilities to make using these permissions simple (for example, `LoginRequiredMixin` and `PermissionRequiredMixin`).

However, it is up to us to use Django's tools appropriately to the job at hand.

A8 Cross Site Request Forgery (CSRF)

CSRF (pronounced *see surf*) is the most technically complex risk in the OWASP Top 10. CSRF relies on the fact that it will automatically send all the cookies associated with the domain whenever a browser requests any resource from a server. A malicious attacker may trick one of our logged in users to view a page on a third-party site (for example, `malicious.example.org`) with, for example, an `img` tag with a `src` attribute that points to a URL from our site (for example, `mymdb.example.com`). When the user's browser sees that `src`, it will make a `GET` request to that URL and send all the cookies (including session ID) associated with our site.

The risk is that if our web app receives a `GET` request, it will make a modification that the user didn't intend. The mitigation for this risk is to make sure that any operation that makes a modification (for example, `UpdateVote`) has a unique and unpredictable value (a CSRF token) that only our system knows, which confirms that the user is intentionally using our app to perform this operation.

Django helps us a lot to mitigate this risk. Django provides the `csrf_token` tag to make it easy to add a CSRF token to a form. Django takes care of adding a matching cookie (to validate against the token) and that any request with a verb that is not a `GET`, `HEAD`, `OPTIONS`, or `TRACE` has a valid CSRF token to be processed. Django further helps us do the right thing by having all its generic editing views (`EditView`, `CreateView`, `DeleteView`, and `FormView`) perform only a modification operation on `POST` and never on `GET`.

However, Django can't save us from ourselves. If we decide to disable this functionality or write views that have side effects on `GET`, Django can't help us.

A9 Using components with known vulnerabilities

A chain is only as strong as its weakest link, and, sometimes, projects can have vulnerabilities in the frameworks and libraries they rely on.

The Django project has a security team that accepts confidential reports of security issues and has a security disclosure policy to keep the community aware of issues affecting their projects. Generally, a Django release receives support (including security updates) for 16 months from its first release, but **Long-Term Support** (**LTS**) releases receive support for 3 years (the next LTS release will be Django 2.2).

However, Django doesn't automatically update itself and doesn't force us to run the latest version. Each deployment must manage this for themselves.

A10 Unvalidated redirects and forwards

If our site can be used to redirect/forward a user to a third-party site automatically, then our site is at risk of having its reputation used to trick users into being forwarded to malicious sites.

Django protects us by making sure that the `next` parameter of `LoginView` will only forward user's URLs that are part of our project.

However, Django can't protect us from ourselves. We have to make sure that we never use use-provided and unvalidated data as the basis of an HTTP redirect or forward.

Summary

In this section, we've updated our app to let users upload images related to movies and reviewed the OWASP Top 10. We covered how Django protects us and also where we need to protect ourselves.

Next, we'll build a list of the top 10 movies and take a look at how to use caching to avoid scanning our entire database each time.

4
Caching in on the Top 10

In this chapter, we'll use the votes that our users have cast to build a list of the top 10 movies in MyMDB. In order to ensure that this popular page remains quick to load, we'll take a look at tools to help us optimize our site. Finally, we'll look at Django's caching API and how to use it to optimize our project.

In this chapter, we will do the following things:

- Create a top 10 movie list using an aggregate query
- Learn about Django instrumentation tools to measure optimization
- Use Django's cache API to cache results of expensive operations

Let's start by making our top 10 movies list page.

Creating a top 10 movies list

For building our top 10 list, we'll start off by creating a new `MovieManager` method and then use it in a new view and template. We'll also update the top header in our base template to make the list easily accessible from every page.

Creating MovieManager.top_movies()

Our `MovieManager` class needs to be able to return a `QuerySet` object of the most popular movies as voted by our users. We're using a naive formula for popularity, that is, the sum of 👍 votes minus the sum of 👎votes. Just like in `Chapter 2`, *Adding Users to MyMDB*, we will use the `QuerySet.annotate()` method to make an aggregate query to count the votes.

Let's add our new method to `django/core/models.py`:

```
from django.db.models.aggregates import (
    Sum
)


class MovieManager(models.Manager):

    # other methods omitted

    def top_movies(self, limit=10):
        qs = self.get_queryset()
        qs = qs.annotate(
            vote_sum=Sum('vote__value'))
        qs = qs.exclude(
            vote_sum=None)
        qs = qs.order_by('-vote_sum')
        qs = qs[:limit]
        return qs
```

We order our results by the sum of their votes (descending) to get our top movies list. However, we face the problem that some movies won't have a vote and so will have `NULL` as their `vote_sum` value. Unfortunately, `NULL` will be ordered first by Postgres. We'll solve this by adding the constraint that a movie with no votes will, by definition, not be one of the top movies. We use `QuerySet.exclude` (which is the opposite of `QuerySet.filter`) to remove movies that don't have a vote.

This is the first time that we see a `QuerySet` object being sliced. A `QuerySet` object is not evaluated by slicing unless a step is provided (for example, `qs [10:20:2]` would make the `QuerySet` object be evaluated immediately and return rows 10, 12, 14, 16, and 18).

Now that we have a `QuerySet` object with the proper `Movie` model instances, we can use the `QuerySet` object in our view.

Creating the TopMovies view

Since our `TopMovies` view will need to show a list, we can use Django's `ListView` like we have before. Let's update `django/core/views.py`:

```
from django.views.generic import ListView
from core.models import Movie

class TopMovies(ListView):
```

```
    template_name = 'core/top_movies_list.html'
    queryset = Movie.objects.top_movies(
        limit=10)
```

Unlike the previous `ListView` classes, we will need to specify a `template_name` attribute. Otherwise, `ListView` would try to use `core/movie_list.html`, which is used by the `MovieList` view.

Next, let's create our template.

Creating the top_movies_list.html template

Our Top 10 Movies page will not need pagination, so the template is pretty simple. Let's create `django/core/templates/core/top_movies_list.html`:

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

{% block title %}
  Top 10 Movies
{% endblock %}

{% block main %}
  <h1 >Top 10 Movies</h1 >
  <ol >
    {% for movie in object_list %}
      <li >
        <a href="{% url "core:MovieDetail" pk=movie.id %}" >
          {{ movie }}
        </a >
      </li >
    {% endfor %}
  </ol >
{% endblock %}
```

Extending `base.html`, we will redefine two template `block` tags. The new `title` template `block` has our new title. The `main` template `block` lists the movies in the `object_list`, including a link to each movie.

Finally, let's update `django/templates/base.html` to include a link to our Top 10 Movies page:

```
{# rest of template omitted #}
<div class="mymdb-masthead">
  <div class="container">
    <nav class="nav">
       {# skipping other nav items #}
       <a
          class="nav-link"
          href="{% url 'core:TopMovies' %}"
        >
        Top 10 Movies
       </a>
       {# skipping other nav items #}
      </nav>
   </div>
</div>
{# rest of template omitted #}
```

Now, let's add a `path()` object to our URLConf so that Django can route requests to our `TopMovies` view.

Adding a path to TopMovies

As always, we will need to add a `path()` to help Django route requests to our view. Let's update `django/core/urls.py`:

```
from django.urls import path

from . import views

app_name = 'core'
urlpatterns = [
    path('movies',
         views.MovieList.as_view(),
         name='MovieList'),
    path('movies/top',
         views.TopMovies.as_view(),
         name="TopMovies"),
    # other paths omitted
 ]
```

With that, we're done. We now have a Top 10 Movies page on MyMDB.

However, looking through all the votes means scanning the largest table in the project. Let's look at ways to optimize our project.

Optimizing Django projects

There is no single correct answer for how to optimize a Django project because different projects have different constraints. To succeed, it's important to be clear about what you're optimizing and what to use in hard numbers, not intuition.

It's important to be clear about what we're optimizing because optimization usually involves trade-offs. Some of the constraints you may wish to optimize for are as follows:

- Response time
- Web server memory
- Web server CPU
- Database memory

Once you know what you're optimizing, you will need a way to measure current performance and the optimized code's performance. Optimized code is often more complex than unoptimized code. You should always confirm that the optimization is effective before taking on the burden of the complexity.

Django is just Python, so you can use a Python profiler to measure performance. This is a useful but complicated technique. Discussing the details of Python profiling goes beyond the scope of this book. However, it's important to remember that Python profiling is a useful tool at our disposal.

Let's take a look at some Django-specific ways that you can measure performance.

Using the Django Debug Toolbar

The Django Debug Toolbar is a third-party package that provides a lot of useful debug information right in the browser. The toolbar is composed of a list of panels. Each panel provides a distinct set of information.

Some of the most useful panels (which are enabled by default) are as follows:

- **Request Panel:** It shows information related to the request, including the view that processed the request, arguments it received (parsed out of the path), cookies, session data, and `GET`/`POST` data in the request.

- **SQL Panel:** It shows how many queries are made, a timeline of their execution, and a button to run `EXPLAIN` on the query. Data-driven web applications are often slowed down by their database queries.
- **Templates Panel:** It shows the templates that were rendered and their context.
- **Logging Panel:** It shows any log messages produced by the view. We'll discuss logging more in the next section.

The profile panel is an advanced panel that is available but not enabled by default. This panel runs a profiler on your view and shows you the results. The panel comes with some caveats, which are explained in the Django Debug Toolbar documentation online (`https://django-debug-toolbar.readthedocs.io/en/stable/panels.html#profiling`).

Django Debug Toolbar is useful in development, but should not be run in production. By default, it will only work if `DEBUG = True` (a setting you must **never** use in production).

Using Logging

Django uses Python's built-in logging system, which you can configure using `settings.LOGGING` . It's configured using a `DictConfig`, as documented in the Python documentation.

As a refresher, here's how Python's logging system works. The system is composed of *loggers*, which receive a *message* and *log level* (for example, `DEBUG` and `INFO`) from our code. If the logger is configured to not filter out messages at that log level (or higher), it creates a *log record* that is passed to all its *handlers*. A handler will check whether it matches the handler's log level, then it will format the log record (using a *formatter*) and emit the message. Different handlers will emit messages differently. `StreamHandler` will write to a stream (`sys.stderr` by default), `SysLogHandler` writes to `SysLog`, and `SMTPHandler` sends an email.

By logging how long operations take, you can get a meaningful sense of what you need to optimize. Using the correct log levels and handlers, you can measure resource consumption in production.

Application Performance Management

Application Performance Management (**APM**) is the name for services that (often) run as part of your application server and trace performed operations. The trace is sent to a reporting server, which combines all the traces, and can give you code line-level insight into the performance of your production servers. This can be helpful for large and complicated deployments, but may be overkill for smaller, simpler web applications.

A quick review of the section

In this section, we reviewed the importance of knowing what to optimize before you actually start optimizing. We also looked at some tools to help us measure whether our optimization was successful.

Next, we'll take a look at how we can solve some common performance problems with Django's cache API.

Using Django's cache API

Django provides a caching API out of the box. In `settings.py`, you can configure one or more caches. Caching can be used to store a whole site, a single page's response, a template fragment, or any pickleable object. Django provides a single API that can be configured with a variety of backends.

In this section, we will perform the following functions:

- Look at the different backends for Django's cache API
- Use Django to cache a page
- Use Django to cache a template fragment
- Use Django to cache a `QuerySet`

One thing we won't be looking at is *downstream* caching, such as **Content Delivery Networks** (**CDNs**) or proxy caches. These are not Django specific, and there is a wide variety of options. Generally speaking, these kinds of caches will rely on the same `VARY` headers that Django has already sent.

Next, let's look at configuring the backends for the cache API.

Examining the trade-offs between Django cache backends

Different backends can be appropriate for different situations. However, the golden rule of caches is that they must be *faster* than the source they're caching or else you've made your application slower. Deciding which backend is appropriate for which task is best done by instrumenting your project, as discussed in the preceding section. Different backends have different trade-offs.

Examining Memcached trade-offs

Memcached is the most popular cache backend, but it still comes with trade-offs that you need to evaluate. Memcached is an in-memory key value store for small data that can be shared by several clients (for example, Django processes) using one or more Memcached hosts. However, Memcached will not be appropriate for caching large blocks of data (1 MB of data, by default). Also, since Memcached is all in-memory, if the process is restarted then the entire cache is cleared. On the other hand, Memcached has remained popular because it is fast and simple.

Django comes with two Memcached backends, depending on the `Memcached` library that you want to use:

- `django.core.cache.backends.memcached.MemcachedCache`
- `django.core.cache.backends.memcached.PyLibMCCache`

You must also install the appropriate library (`python-memcached` or `pylibmc`, respectively). To specify the address(es) of your Memcached servers set `LOCATION` to a list in the format `address:PORT` (for example, `['memcached.example.com:11211',]`). An example configuration is listed at the end of this section.

Using Memcached in *development* and *testing* is unlikely to be very useful, unless you have evidence to the contrary (for example, you need to replicate a complex bug).

Memcached is popular in production environments because it is fast and easy to set up. It avoids duplication of data by letting all your Django processes connect to the same host(s). However, it uses a lot of memory (and degrades quickly and poorly when it runs out of available memory). It's also important to be mindful of the operational costs of running another service.

Here's an example config for using `memcached`:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache',
        'LOCATION':  [
            '127.0.0.1:11211',
        ],
    }
}
```

Examining dummy cache trade-offs

The **dummy cache** (`django.core.cache.backends.dummy.DummyCache`) will check whether a key is valid, but otherwise will perform no operations.

This cache can be useful for *development* and *testing* when you want to make sure that you're definitely seeing the results of your code changes, not something cached.

Don't use this cache in *production*, as it has no effect.

Here's an example config for the dummy cache:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

Examining local memory cache trade-offs

The **local memory cache** (`django.core.cache.backends.locmem.LocMemCache`) uses a Python dictionary as a global in-memory cache. If you want to use multiple separate local memory caches, give each unique string in `LOCATION`. It's called a local cache because it's local to each process. If you're spinning up multiple processes (as you would in production), then you might cache the same value multiple times as different processes handle requests. This inefficiency may be preferable for its simplicity, as it does not require another service.

This is a useful cache to use in *development* and *testing* to confirm that your code is caching correctly.

You may want to use this in *production*, but keep in mind the potential inefficiency of different processes caching the same data.

The following is an example config for the local memory cache:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'defaultcache',

    },
    'otherCache': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'othercache',
    }
}
```

Examine file-based cache trade-offs

Django's **file-based cache** (`django.core.cache.backends.filebased.FileBasedCache`) uses compressed files in a specified `LOCATION` directory to cache data. Using files may seem strange; aren't caches supposed to be *fast* and files *slow*? The answer, again, depends on what you're caching. As an example, network requests to an external API may be slower than your local disk. Remember that each server will have a separate disk, so there will be some duplication of data if you're running a cluster.

You probably don't want to use this in *development* or *testing* unless you are heavily memory constrained.

You may want to use this in production to cache resources that are particularly large or slow to request. Remember that you should give your server's process write permission to the `LOCATION` directory. Also, make sure that you give your server(s) enough disk space for your cache.

The following is an example config to use the file-based cache:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': os.path.join(BASE_DIR, '../file_cache'),
    }
}
```

Examining database cache trade-offs

The **database cache** backend (`django.core.cache.backends.db.DatabaseCache`) uses a database table (named in `LOCATION`) to store the cache. Obviously, this works best if your database is fast. Depending on the scenario, this may be helpful even when caching results of database queries if the queries are complex but single row lookups are fast. There are upsides to this, as the cache is not ephemeral like a memory cache and can be easily shared across processes and servers (such as Memcached).

The database cache table is not managed by a migration but by a `manage.py` command, as follows:

```
$ cd django
$ python manage.py createcachetable
```

You probably don't want to use this in *development* or *testing* unless you want to replicate your production environment locally.

You may want to use this in *production* if your testing proves that it's appropriate. Remember to consider what the increased database load will do to its performance.

The following is an example config to use the database cache:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'django_cache_table',
    }
}
```

Configuring a local memory cache

In our case, we will use a local memory cache with a very low timeout. This will mean that most requests we make while writing our code will skip the cache (old values, if any, will have expired), but if we quickly click on refresh, we'll be able to get confirmation that our cache is working.

Let's update `django/config/settings.py` to use a local memory cache:

```
CACHES = {
   'default': {
       'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
       'LOCATION': 'default-locmemcache',
       'TIMEOUT': 5, # 5 seconds
   }
}
```

Although we can have multiple differently configured caches, the default cache is expected to be named `'default'`.

`Timeout` is how long (in seconds) a value should be kept in the cache before it's culled (removed/ignored). If `Timeout` is `None`, then the value will be considered to never expire.

Now that we have a cache configured, let's cache the `MovieList` page.

Caching the movie list page

We will proceed on the assumption that the `MovieList` page is very popular and expensive for us generate. To reduce the cost of serving these requests, we will use Django to cache the entire page.

Django provides the decorator (function) `django.views.decorators.cache.cache_page`, which can be used to cache a single page. It may seem strange that this is a decorator instead of a mixin. When Django was initially launched, it didn't have **Class-Based Views** (**CBVs**), only **Function-Based Views** (**FBVs**). As Django matured, much of the code switched to using CBVs, but there are still some features implemented as FBV decorators.

There are a few different ways to use function decorators in CBVs. Our approach will be to build our own mixin. Much of the power of CBVs comes off from the power of being able to mix in new behavior to existing classes. Knowing how to do that is a useful skill.

Creating our first mixin – CachePageVaryOnCookieMixin

Let's create a new class in `django/core/mixins.py`:

```
from django.core.cache import caches
from django.views.decorators.cache import (
    cache_page)


class CachePageVaryOnCookieMixin:
    """
    Mixin caching a single page.

    Subclasses can provide these attributes:

    `cache_name` - name of cache to use.
    `timeout` - cache timeout for this
    page. When not provided, the default
    cache timeout is used.
    """
    cache_name = 'default'

    @classmethod
    def get_timeout(cls):
        if hasattr(cls, 'timeout'):
            return cls.timeout
        cache = caches[cls.cache_name]
        return cache.default_timeout

    @classmethod
    def as_view(cls, *args, **kwargs):
        view = super().as_view(
            *args, **kwargs)
        view = vary_on_cookie(view)
        view = cache_page(
            timeout=cls.get_timeout(),
            cache=cls.cache_name,
        )(view)
        return view
```

Our new mixin overrides the `as_view()` class method that we use in URLConfs and decorates the view with the `vary_on_cookie()` and `cache_page()` decorators. This effectively acts as if we were decorating the `as_view()` method with our function decorator.

Let's look at the `cache_page()` decorator first. `cache_page()` requires a `timeout` argument and optionally takes a `cache` argument. `timeout` is how long (in seconds) before the cached page should expire and must be recached. Our default timeout value is the default for the cache we're using. Classes that subclass `CachePageVaryOnCookieMixin` can provide a new `timeout` attribute just like our `MovieList` class provides a `model` attribute. The `cache` argument expects the string name of the desired cache. Our mixin is set up to use the `default` cache, but by referencing it via a class attribute, that too can be changed by a subclass.

When caching a page such as `MoveList`, we must remember that the resulting page is different for different users. In our case, the header of `MovieList` looks different for logged in users (shows a *log out* link) and for logged out users (shows *log in* and *register* links). Django, again, does the heavy work for us by providing the `vary_on_cookie()` decorator.

The `vary_on_cookie()` decorator adds a `VARY cookie` header to the response. The `VARY` header is used by caches (both downstream and Django's) to let them know about variants of that resource. `VARY cookie` tells the cache that each different cookie/URL pair is a different resource and should be cached separately. This means that logged in users and logged out users will not see the same page because they will have different cookies.

This has an important impact on our hit ratio (the proportion of times a cache is *hit* instead of the resource being regenerated). A cache with a low hit ratio will have minimal effect, as most requests will *miss* the cache and result in a processed request.

In our case, we also use cookies for CSRF protection. While session cookies may lower a hit ratio a bit, depending on the circumstance (look at your user's activity to confirm), a CSRF cookie is practically fatal. The nature of a CSRF cookie is to change a lot so that attackers cannot predict it. If that constantly changing value is sent with many requests, then very few can be cached. Luckily, we can move our CSRF value out of cookies and into the server side session with a `settings.py` change.

Deciding on the right CSRF strategy for your app can be complex. For example, AJAX applications will want to add CSRF tokens through headers. For most sites, the default Django configuration (using cookies) is fine. If you need to change it, it's worth reviewing Django's CSRF protection documentation (`https://docs.djangoproject.com/en/2.0/ref/csrf/`).

In `django/conf/settings.py`, add the following code:

```
CSRF_USE_SESSIONS = True
```

Now, Django won't send the CSRF token in a cookie, but will store it in the user's session (stored on the server).

If users already have CSRF cookies, they will be ignored; however, it will still have a dampening effect on the hit ratio. In production, you may wish to consider adding a bit of code to delete those CSRF cookies.

Now that we have a way of easily mixing in caching behavior, let's use it in our `MovieList` view.

Using CachePageVaryOnCookieMixin with MovieList

Let's update our view in `django/core/views.py`:

```
from django.views.generic import ListView
from core.mixins import (
    VaryCacheOnCookieMixin)


class MovieList(VaryCacheOnCookieMixin, ListView):
    model = Movie
    paginate_by = 10

    def get_context_data(self, **kwargs):
        # omitted due to no change
```

Now when `MovieList` gets a request routed to it, `cache_page` will check whether it has already been cached. If it has been cached, Django will return the cached response without doing any more work. If it hasn't been cached, our regular `MovieList` view will create a new response. The new response will have a `VARY cookie` header added and then get cached.

Next, let's try to cache a part of our Top 10 movie list inside a template.

Caching a template fragment with {% cache %}

Sometimes, pages load slowly because a part of our template is slow. In this section, we'll take a look at how to solve this problem by caching a fragment of our template. For example, if you are using a tag that takes a long time to resolve (say, because it makes a network request), then it will slow down any page that uses that tag. If you can't optimize the tag itself, it may be sufficient to cache its result in the template.

Let's cache our rendered Top 10 Movies list by editing `django/core/templates/core/top_movies.html`:

```
{% extends "base.html" %}
{% load cache %}

{% block title %}
  Top 10 Movies
{% endblock %}

{% block main %}
  <h1 >Top 10 Movies</h1 >
  {% cache 300 top10 %}
  <ol >
    {% for movie in object_list %}
      <li >
        <a href="{% url "core:MovieDetail" pk=movie.id %}" >
          {{ movie }}
        </a >
      </li >
    {% endfor %}
  </ol >
  {% endcache %}
{% endblock %}
```

This block introduces us to the `{% load %}` tag and the `{% cache %}` tag.

The `{% load %}` tag is used to load a library of tags and filters and make them available for use in a template. A library may provide one or more tags and/or filters. For example, `{% load humanize %}` loads tags and filters to make values look more human. In our case, `{% load cache %}` provides only the `{% cache %}` tag.

`{% cache 300 top10 %}` will cache the body of the tag for the provided number of seconds under the provided key. The second argument must be a hardcoded string (not a variable), but we can provide more arguments if the fragment needs to have variants (for example, `{% cache 300 mykey request.user.id %}` to cache a separate fragment for each user). The tag will use the `default` cache unless the last argument is `using='cachename'`, in which case the named cache will be used instead.

Caching with `{% cache %}` happens at a different level than when using `cache_page` and `vary_on_cookie`. All the code in the view will still be executed. Any slow code in the view will still slow us down. Caching a template fragment solves only one very particular case of a slow fragment in our template code.

Since `QuerySets` are lazy by putting our `for` loop inside `{% cache %}`, we've avoided evaluating the `QuerySet`. If we want to cache a value to avoid querying it, our code would be much clearer if we did it in the view.

Next, let's look at how to cache an object using Django's cache API.

Using the cache API with objects

The most granular use of Django's cache API is to store objects compatible with Python's `pickle` serialization module. The `cache.get()`/`cache.set()` methods we'll see here are used internally by the `cache_page()` decorator and the `{% cache %}` tag. In this section, we'll use these methods to cache the `QuerySet` returned by `Movie.objects.top_movies()`.

Conveniently, `QuerySet` objects are pickleable. When a `QuerySets` is pickled, it will immediately be evaluated, and the resulting models will be stored in the built-in cache of the `QuerySet`. When unpickling a `QuerySet`, we can iterate over it without causing new queries. If the `QuerySet` had `select_related` or `prefetch_related`, those queries would execute on pickling and *not* rerun on unpickling.

Let's remove our `{% cache %}` tag from `top_movies_list.html` and instead update `django/core/views.py`:

```
import django
from django.core.cache import cache
from django.views.generic import ListView

from core.models import Movie


class TopMovies(ListView):
    template_name = 'core/top_movies_list.html'

    def get_queryset(self):
        limit = 10
        key = 'top_movies_%s' % limit
        cached_qs = cache.get(key)
        if cached_qs:
```

```
            same_django = cached_qs._django_version == django.get_version()
            if same_django:
                return cached_qs
        qs = Movie.objects.top_movies(
            limit=limit)
        cache.set(key, qs)
        return qs
```

Our new `TopMovies` view overrides the `get_queryset` method and checks the cache before using `MovieManger.top_movies()`. Pickling `QuerySet` objects does come with one caveat—they are not guaranteed to be compatible across Django versions, so we should check the version used before proceeding.

`TopMovies` also shows a different way of accessing the default cache than what `VaryOnCookieCache` used. Here, we import and use `django.core.cache.cache`, which is a proxy for `django.core.cache.caches['default']`.

It's important to remember the importance of consistent keys when caching using a low-level API. In a large code base, it's easy to store the same data under different keys leading to inefficiency. It can be convenient to put the caching code into your manager or into a utility module.

Summary

In this chapter, we made a Top 10 Movies view, reviewed tools for instrumenting your Django code, and covered how to use Django's cache API. Django and Django's community provide tools for helping you discover where to optimize your code using profilers, the Django Debug Toolbar, and logging. Django's caching API helps us with a rich API to cache whole pages with `cache_page`, the `{% cache %}` template tag for template fragments, and `cache.set`/`cache.get` for caching any picklable object.

Next, we'll deploy MyMDB with Docker.

5
Deploying with Docker

In this chapter, we'll look at how to deploy MyMDB into a production environment using Docker containers hosted on a Linux server in Amazon's **Electric Computing Cloud** (**EC2**). We will also use **Simple Storage Service** (**S3**) of **Amazon Web Services** (**AWS**) to store files that users upload.

We will do the following things:

- Split up our requirements and settings files to separate development and production settings
- Build a Docker container for MyMDB
- Build a database container
- Use Docker Compose to launch both containers
- Launch MyMDB into a production environment on a Linux server in the cloud

First, let's split up our requirements and settings so that our development and production values are kept separate.

Organizing configuration for production and development

Till now, we've kept a single requirements file and a single `settings.py` file. This has made development convenient. However, we can't use our development settings in production.

The current best practice is to have a separate file for each environment. Each environment's file then imports a common file with shared values. We'll use this pattern for requirements and settings files.

Let's start by splitting up our requirements files.

Splitting requirements files

Let's create `requirements.common.txt` at the root of our project:

```
django<2.1
psycopg2
Pillow<4.4.0
```

Regardless of the environment that we're in, we always need Django, Postgres drivers, and Pillow (for the `ImageField` class). However, this requirements file is never used directly.

Next, let's list our development requirements in `requirements.dev.txt`:

```
-r requirements.common.txt
django-debug-toolbar==1.8
```

The preceding file will install everything from `requirements.common.txt` (thanks to `-r`) and the Django Debug Toolbar.

For our production packages, we'll use `requirements.production.txt`:

```
-r requirements.common.txt
django-storages==1.6.5
boto3==1.4.7
uwsgi==2.0.15
```

This will also install the packages from `requirements.common.txt`. It will also install the `boto3` and `django-storages` packages to help us upload files to S3 easily. The `uwsgi` package will provide the server we'll use to serve Django.

To install packages for production, we can now execute the following command:

```
$ pip install -r requirements.production.txt
```

Next, let's split up the settings file along similar lines.

Splitting settings file

Again, we will follow the current Django best practice of splitting our settings file into the following three files: `common_settings.py`, `production_settings.py`, and `dev_settings.py`.

Creating common_settings.py

We'll create `common_settings.py` by renaming our current `settings.py` file and then making the changes mentioned in this section.

Let's change `DEBUG = False` so that no new settings file can *accidentally* be in debug mode. Then, let's change the `SECRET_KEY` setting to get its value from an environment variable, by changing its line to be:

```
SECRET_KEY = os.getenv('DJANGO_SECRET_KEY')
```

Let's also add a new setting, `STATIC_ROOT`. `STATIC_ROOT` is the directory where Django will collect all the static files from across our installed apps to make it easier to serve them:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'gathered_static_files')
```

In the database config, we can remove all the credentials but keep the `ENGINE` value (to make it clear, we intend to use Postgres everywhere):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
    }
}
```

Finally, let's delete the `CACHES` setting. This will have to be configured differently in each environment.

Next, let's create a development settings file.

Creating dev_settings.py

Our development settings will be in `django/config/dev_settings.py`. We'll build it incrementally.

First, we will import everything from `common_settings`:

```
from config.common_settings import *
```

Then, we'll override the `DEBUG` and `SECRET_KEY` settings:

```
DEBUG = True
SECRET_KEY = 'some secret'
```

In development, we want to run in debug mode. We will also feel safe hardcoding a secret key, as we know that it won't be used in production.

Next, let's update the `INSTALLED_APPS` list:

```
INSTALLED_APPS += [
    'debug_toolbar',
]
```

In development, we can run extra apps (such as the Django Debug Toolbar) by appending a list of development-only apps to the `INSTALLED_APPS` list.

Then, let's update the database configuration:

```
DATABASES['default'].update({
    'NAME': 'mymdb',
    'USER': 'mymdb',
    'PASSWORD': 'development',
    'HOST': 'localhost',
    'PORT': '5432',
})
```

Since our development database is local, we can hardcode the values in our settings to make the file simpler. If your database is not local, avoid checking passwords into version control and use `os.getenv()`, as in production.

Next, let's update the cache configuration:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'default-locmemcache',
        'TIMEOUT': 5,
    }
}
```

We'll use a very short timeout in our development cache.

Finally, we need to set file upload directory:

```
# file uploads
MEDIA_ROOT = os.path.join(BASE_DIR, '../media_root')
```

In development, we'll store uploaded files on our local filesystem in development. We will specify the directory to upload to using `MEDIA_ROOT`.

The Django Debug Toolbar needs a bit of configuration as well:

```
# Django Debug Toolbar
INTERNAL_IPS = [
    '127.0.0.1',
]
```

The Django Debug Toolbar will only render at predefined IPs, so we will give it our localhost IP so that we can use it locally.

We can also add more settings that our development-only apps may require.

Next, let's add production settings.

Creating production_settings.py

Let's create our production settings in `django/config/production_settings.py`.

`production_settings.py` is similar to `dev_settings.py` but often uses `os.getenv()` to get values from environment variables. This helps us keep secrets (for example, Passwords, API tokens, and so on) out of version control and decouples settings from particular servers:

```
from config.common_settings import *
DEBUG = False
assert SECRET_KEY is not None, (
    'Please provide DJANGO_SECRET_KEY '
    'environment variable with a value')
ALLOWED_HOSTS += [
    os.getenv('DJANGO_ALLOWED_HOSTS'),
]
```

First, we import the common settings. Out of an abundance of caution, we ensure that the debug mode is off.

Having a `SECRET_KEY` set is vital to our system staying secure. We `assert` to prevent Django from starting up without `SECRET_KEY`. The `common_settings` module should have already set it from an environment variable.

A production website will be accessed from a domain other than `localhost`. We then tell Django what other domains we're serving by appending the `DJANGO_ALLOWED_HOSTS` environment variable to the `ALLOWED_HOSTS` list.

Next, we'll update the database configuration:

```
DATABASES['default'].update({
    'NAME': os.getenv('DJANGO_DB_NAME'),
    'USER': os.getenv('DJANGO_DB_USER'),
    'PASSWORD': os.getenv('DJANGO_DB_PASSWORD'),
    'HOST': os.getenv('DJANGO_DB_HOST'),
    'PORT': os.getenv('DJANGO_DB_PORT'),
})
```

We update the database configuration using values from environment variables.

Then, the cache configuration needs to be set.

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'default-locmemcache',
        'TIMEOUT': int(os.getenv('DJANGO_CACHE_TIMEOUT'), ),
    }
}
```

In production, we will accept the trade-offs of a local memory cache. We configure the timeout at runtime using another environment variable.

Next, the file upload configuration settings need to bedded.

```
# file uploads
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
AWS_ACCESS_KEY_ID = os.getenv('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.getenv('AWS_SECRET_ACCESS_KEY_ID')
AWS_STORAGE_BUCKET_NAME = os.getenv('DJANGO_UPLOAD_S3_BUCKET')
```

In production, we won't store uploaded images on our container's local filesystem. One core concept of Docker is that containers are ephemeral. It should be acceptable to stop and delete a container and replace it with another. If we stored uploaded images locally, we'd go against that philosophy.

Another reason for not storing uploaded files locally is that they should also be served from a different domain (we discussed this in `Chapter 3`, *Posters, Headshots, and Security*). We will use S3 storage since it's cheap and easy.

The `django-storages` app provides file storage backends for many CDNs, including S3. We tell Django to use that S3 by changing the `DEFAULT_FILE_STORAGE` setting.
The `S3Boto3Storage` backend requires a few more settings to be able to work with AWS, including an AWS Access Key, an AWS Secret Access Key, and the name of the destination bucket. We'll discuss the two Access Keys later, in the AWS section.

Now that our settings are organized, we can create our MyMDB `Dockerfile`.

Creating the MyMDB Dockerfile

In this section, we will create a Dockerfile for MyMDB. Docker runs containers based on an image. An image is defined by a Dockerfile. A Dockerfile must extend another Dockerfile (the reserved `scratch` image being the end of this cycle).

Docker's philosophy is that each container should have a single concern (purpose). This may mean that it runs a single process, or it may run multiple processes working together. In our case, it will run both uWSGI and Nginx processes to provide MyMDB.

Confusingly, Dockerfile refers to both the expected *filename* and the *file type*. So `Dockerfile` is a Dockerfile.

Let's create a Dockerfile at the root of our project in a file called `Dockerfile`. Dockerfile uses its own language to define the files/directories in the image, as well as any commands required to run while making the image. A complete guide on writing a Dockerfile is out of the scope of this chapter. Instead, we'll build our `Dockerfile` incrementally, discussing only the most relevant elements.

We'll build our `Dockerfile` by following six steps:

1. Initializing the base image and adding the source code to the image
2. Installing packages
3. Collecting static files
4. Configuring Nginx
5. Configuring uWSGI
6. Cleaning up unnecessary resources

Starting our Dockerfile

The first part of our `Dockerfile` tells Docker which image to use as the base, adds our code, and creates some common directories:

```
FROM phusion/baseimage

# add code and directories
RUN mkdir /mymdb
WORKDIR /mymdb
COPY requirements* /mymdb/
COPY django/ /mymdb/django
COPY scripts/ /mymdb/scripts
RUN mkdir /var/log/mymdb/
RUN touch /var/log/mymdb/mymdb.log
```

Let's look at these instructions in more detail:

- `FROM`: This is required in a Dockerfile. `FROM` tells Docker what image to use as the base image for our image. We will use `phusion/baseimage` because it provides a lot of convenient facilities and uses very little memory. It's a tailored-for-Docker Ubuntu image with a smaller easy-to-use init service manager called runit (instead of the Ubuntu's upstart).
- `RUN`: This executes a command as part of building the image. `RUN mkdir /mymdb` creates the directory in which we'll store our files.
- `WORKDIR`: This sets the working directory for all our future `RUN` commands.
- `COPY`: This adds a file (or directory) from our filesystem to the image. Source paths are relative to the directory containing our `Dockerfile`. It's best to make the destination path an absolute path.

We will also reference a new directory called `scripts`. Let's create it at the root of our project directory:

```
$ mkdir scripts
```

As part of configuring and building the new image, we'll create a few small bash scripts that we'll keep in the `scripts` directory.

Installing packages in Dockerfile

Next, we'll tell our `Dockerfile` to install all the packages we will need:

```
RUN apt-get -y update
RUN apt-get install -y \
    nginx \
    postgresql-client \
    python3 \
    python3-pip
RUN pip3 install virtualenv
RUN virtualenv /mymdb/venv
RUN bash /mymdb/scripts/pip_install.sh /mymdb
```

We used `RUN` statements to install the Ubuntu packages and create a virtual environment. To install our Python packages into our virtual environment, we'll create a small script in `scripts/pip_install.sh`:

```
#!/usr/bin/env bash

root=$1
source $root/venv/bin/activate

pip3 install -r $root/requirements.production.txt
```

The preceding script simply activates the virtual environment and runs `pip3 install` on our production requirements file.

It's often hard to debug long commands in the middle of a Docker file. Wrapping commands in scripts can make them easier to debug. If something isn't working, you can connect to a container using the `docker exec -it bash -l` command and debug the script as normal.

Collecting static files in Dockerfile

Static files are the CSS, JavaScript, and images that support our website. Static files may not always be created by us. Some static files come from installed Django apps (for example, Django admin). Let's update our `Dockerfile` to collect the static files:

```
# collect the static files
RUN bash /mymdb/scripts/collect_static.sh /mymdb
```

Again, we've wrapped the command in a script. Let's add the following script to `scripts/collect_static.sh`:

```
#!/usr/bin/env bash

root=$1
source $root/venv/bin/activate

export DJANGO_CACHE_TIMEOUT=100
export DJANGO_SECRET_KEY=FAKE_KEY
export DJANGO_SETTINGS_MODULE=config.production_settings

cd $root/django/

python manage.py collectstatic
```

The preceding script activates the virtual environment we created in the preceding code and sets the required environment variables. Most of these values don't matter in this context as long as the variables are present. However,
the `DJANGO_SETTINGS_MODULE` environment variable is very important.
The `DJANGO_SETTINGS_MODULE` environment variable is used by Django to find the settings module. If we don't set it and don't have `config/settings.py`, then Django won't start (even `manage.py` commands will fail).

Adding Nginx to Dockerfile

To configure Nginx, we will add a config file and a runit service script:

```
COPY nginx/mymdb.conf /etc/nginx/sites-available/mymdb.conf
RUN rm /etc/nginx/sites-enabled/*
RUN ln -s /etc/nginx/sites-available/mymdb.conf /etc/nginx/sites-
enabled/mymdb.conf

COPY runit/nginx /etc/service/nginx
RUN chmod +x /etc/service/nginx/run
```

Configuring Nginx

Let's add an Nginx configuration file to `nginx/mymdb.conf`:

```
# the upstream component nginx needs
# to connect to
upstream django {
    server 127.0.0.1:3031;
}

# configuration of the server
server {

    # listen on all IPs on port 80
    server_name 0.0.0.0;
    listen      80;
    charset     utf-8;

    # max upload size
    client_max_body_size 2M;

    location /static {
        alias /mymdb/django/gathered_static_files;
    }

    location / {
        uwsgi_pass  django;
        include     /etc/nginx/uwsgi_params;
    }

}
```

Nginx will be responsible for the following two things:

- Serving static files (URLs starting with `/static`)
- Passing all other requests to uWSGI

The `upstream` block describes the location of our Django (uWSGI) server. In the `location / ` block, nginx is instructed to pass requests on to the upstream server using the uWSGI protocol. The `include /etc/nginx/uwsgi_params` file describes how to map headers so that uWSGI understands them.

`client_max_body_size` is an important setting. It describes the maximum size for file uploads. Leaving this value too big can expose a vulnerability, as attackers can overwhelm the server with huge requests.

Creating Nginx runit service

In order for `runit` to know how to start Nginx, we will need to provide a `run` script. Our `Dockerfile` expects it to be in `runit/nginx/run`:

```
#!/usr/bin/env bash

exec /usr/sbin/nginx \
    -c /etc/nginx/nginx.conf \
    -g "daemon off;"
```

`runit` doesn't want its services to fork off a separate process, so we run Nginx with `daemon off`. Further, `runit` wants us to use `exec` to replace our script's process, the new Nginx process.

Adding uWSGI to the Dockerfile

We're using uWSGI because it often ranks as the fastest WSGI app server. Let's set it up in our `Dockerfile` by adding the following code:

```
# configure uwsgi
COPY uwsgi/mymdb.ini /etc/uwsgi/apps-enabled/mymdb.ini
RUN mkdir -p /var/log/uwsgi/
RUN touch /var/log/uwsgi/mymdb.log
RUN chown www-data /var/log/uwsgi/mymdb.log
RUN chown www-data /var/log/mymdb/mymdb.log

COPY runit/uwsgi /etc/service/uwsgi
RUN chmod +x /etc/service/uwsgi/run
```

This instructs Docker to use a `mymdb.ini` file to configure uWSGI, creates log directories, and adds a uWSGI runit service. In order for runit to start the uWSGI service, we give the runit script permission to execute using the `chmod` command.

Configuring uWSGI to run MyMDB

Let's create the uWSGI configuration in `uwsgi/mymdb.ini`:

```
[uwsgi]
socket = 127.0.0.1:3031
chdir = /mymdb/django/
virtualenv = /mymdb/venv
wsgi-file = config/wsgi.py
```

```
env = DJANGO_SECRET_KEY=$(DJANGO_SECRET_KEY)
env = DJANGO_LOG_LEVEL=$(DJANGO_LOG_LEVEL)
env = DJANGO_ALLOWED_HOSTS=$(DJANGO_ALLOWED_HOSTS)
env = DJANGO_DB_NAME=$(DJANGO_DB_NAME)
env = DJANGO_DB_USER=$(DJANGO_DB_USER)
env = DJANGO_DB_PASSWORD=$(DJANGO_DB_PASSWORD)
env = DJANGO_DB_HOST=$(DJANGO_DB_HOST)
env = DJANGO_DB_PORT=$(DJANGO_DB_PORT)
env = DJANGO_CACHE_TIMEOUT=$(DJANGO_CACHE_TIMEOUT)
env = AWS_ACCESS_KEY_ID=$(AWS_ACCESS_KEY_ID)
env = AWS_SECRET_ACCESS_KEY_ID=$(AWS_SECRET_ACCESS_KEY_ID)
env = DJANGO_UPLOAD_S3_BUCKET=$(DJANGO_UPLOAD_S3_BUCKET)
env = DJANGO_LOG_FILE=$(DJANGO_LOG_FILE)
processes = 4
threads = 4
```

Let's take a closer look at some of these settings:

- `socket` tells uWSGI to open a socket on `127.0.0.1:3031` using its custom `uwsgi` protocol (confusingly, the protocol and the server have the same name).
- `chdir` changes the processes's working directory. All paths need to be relative to this location.
- `virtualenv` tells uWSGI the path to the project's virtual environment.
- Each `env` instruction sets an environment variable for our process. We can access these with `os.getenv()` in our code (for example, `production_settings.py`).
- `$(...)` are references environment variables from the uWSGI process's own environment (for example, `$(DJANGO_SECRET_KEY )`).
- `proccesses` sets how many processes we should run.
- `threads` sets how many threads each process should have.

The `processes` and `threads` settings will need to be fine-tuned based on production performance.

Creating the uWSGI runit service

In order for runit to know how to start uWSGI, we will need to provide a `run` script. Our `Dockerfile` expects it to be in `runit/uwsgi/run`. This script is more complex than what we used for Nginx:

```
#!/usr/bin/env bash

source /mymdb/venv/bin/activate
```

```
export PGPASSWORD="$DJANGO_DB_PASSWORD"
psql \
    -h "$DJANGO_DB_HOST" \
    -p "$DJANGO_DB_PORT" \
    -U "$DJANGO_DB_USER" \
    -d "$DJANGO_DB_NAME"

if [[ $? != 0 ]]; then
    echo "no db server"
    exit 1
fi

pushd /mymdb/django

python manage.py migrate

if [[ $? != 0 ]]; then
    echo "can't migrate"
    exit 2
fi
popd

exec /sbin/setuser www-data \
    uwsgi \
    --ini /etc/uwsgi/apps-enabled/mymdb.ini \
    >> /var/log/uwsgi/mymdb.log \
    2>&1
```

This script does the following three things:

- Checks whether it can connect to the DB, exiting otherwise
- Runs all the migrations or exits on failure
- Starts uWSGI

runit requires that we use `exec` to start our process so that uWSGI will replace the `run` script's process.

Finishing our Dockerfile

As the final step, we will clean up and document the port we're using:

```
RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

EXPOSE 80
```

The `EXPOSE` statement documents which port we're using. Importantly, it does not actually open any ports. We'll have to do that when we run the container.

Next, let's create a container for our database.

Creating a database container

We will need a database to run Django in production. The PostgreSQL Docker community provides us with a very robust Postgres image that we can extend.

Let's create another container for our database in `docker/psql/Dockerfile`:

```
FROM postgres:10.1

ADD make_database.sh /docker-entrypoint-initdb.d/make_database.sh
```

The base image for this `Dockerfile` will use Postgres 10.1. It also has a convenient facility that it will execute any shell or SQL scripts in `/docker-entrypoint-initdb.d` as part of the DB initialization. We'll take advantage of this to create our MyMDB database and user.

Let's create our database initialization script in `docker/psql/make_database.sh`:

```
#!/usr/bin/env bash

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
    CREATE DATABASE $DJANGO_DB_NAME;
    CREATE USER $DJANGO_DB_USER;
    GRANT ALL ON DATABASE $DJANGO_DB_NAME TO "$DJANGO_DB_USER";
    ALTER USER $DJANGO_DB_USER PASSWORD '$DJANGO_DB_PASSWORD';
    ALTER USER $DJANGO_DB_USER CREATEDB;
EOSQL
```

We used a shell script in the preceding code so that we can use environment variables to populate our SQL.

Now that we have both our containers ready, let's make sure that we can actually launch them by signing up for and configuring AWS.

Storing uploaded files on AWS S3

We expect our MyMDB to save files to S3. To accomplish that, we will need to sign up for AWS and then configure our shell to be able to use AWS.

Signing up for AWS

To sign up, navigate to `https://aws.amazon.com` and follow their instructions. Note that signing up is free.

The resources we will use are all in the AWS free tier at the time of writing this book. Some elements of the free tier are only available to new accounts for the first year. Review your account's eligibility before executing any AWS command.

Setting up the AWS environment

To interact with the AWS API, we will need the following two tokens—an Access Key and a Secret Access Key. This key pair defines access to an account.

To generate a pair of tokens, go to `https://console.aws.amazon.com/iam/home?region=us-west-2#/security_credential_`, click on **Access Keys**, and then click on the **create new access keys** button. There is no way to retrieve a **Secret Access Key** if you lose it, so ensure that you save it in a safe place.

The preceding AWS Console link will generate tokens for your root account. This is fine while we're testing things out. In future, you should make users with limited permissions using the AWS IAM permissions system.

Next, let's install the AWS **command-line interface** (**CLI**):

```
$ pip install awscli
```

Then,we need to configure the AWS command line tool with our key and region. The `aws` command offers an interactive `configure` subcommand to do this. Let's run it on the command line:

```
$ aws configure
 AWS Access Key ID [None]: <Your ACCESS key>
 AWS Secret Access Key [None]: <Your secret key>
 Default region name [None]: us-west-2
 Default output format [None]: json
```

The `aws configure` command stores the values you entered in a `.aws` directory in your home directory.

To confirm that your new account is set up correctly, request a list of EC2 instances (there should be none):

```
$ aws ec2 describe-instances
{
    "Reservations": []
}
```

Creating the file upload bucket

S3 is organized into buckets. Each bucket must have a unique name (unique across all of AWS). Each bucket will also have a policy, which controls access.

Let's create a bucket for our file uploads by executing the following commands (change `BUCKET_NAME` to your own unique name):

```
$ export AWS_ACCESS_KEY=#your value
$ export AWS_SECRET_ACCESS_KEY=#yourvalue
$ aws s3 mb s3://BUCKET_NAME
```

To let unauthenticated users access the files in our bucket, we must set a policy. Let's create the policy in `AWS/mymdb-bucket-policy.json`:

```
{
    "Version": "2012-10-17",
    "Id": "mymdb-bucket-policy",
    "Statement": [
        {
            "Sid": "allow-file-download-stmt",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::BUCKET_NAME/*"
        }
    ]
}
```

Ensure that you update `BUCKET_NAME` to the name of your bucket.

Now, we can apply the policy on your bucket using the AWS CLI:

```
$ aws s3api put-bucket-policy --bucket BUCKET_NAME --policy "$(cat
AWS/mymdb-bucket-policy.json)"
```

Ensure that you remember your bucket name, AWS access key, and AWS secret access key as we'll use them in the next section.

Using Docker Compose

We now have all the pieces of production deployment ready. Docker Compose is how Docker lets multiple containers work together. Docker Compose is made of a command-line tool, `docker-compose`; a configuration file, `docker-compose.yml`; and an environment variable file, `.env`. We will create both these files at the root of our project directory.

Never check your `.env` file into version control. That's where your secrets live. Don't let them leak.

First, let's list our environment variables in `.env`:

```
# Django settings
DJANGO_SETTINGS_MODULE=config.production_settings
DJANGO_SECRET_KEY=#put your secret key here
DJANGO_LOG_LEVEL=DEBUG
DJANGO_LOG_FILE=/var/log/mymdb/mymdb.log
DJANGO_ALLOWED_HOSTS=# put your domain here
DJANGO_DB_NAME=mymdb
DJANGO_DB_USER=mymdb
DJANGO_DB_PASSWORD=#put your password here
DJANGO_DB_HOST=db
DJANGO_DB_PORT=5432
DJANGO_CACHE_TIMEOUT=200

AWS_ACCESS_KEY_ID=# put aws key here
AWS_SECRET_ACCESS_KEY_ID=# put your secret key here
DJANGO_UPLOAD_S3_BUCKET=# put BUCKET_NAME here

# Postgres settings
POSTGRES_PASSWORD=# put your postgress admin password here
```

Many of these values are okay to hardcode, but there are a few values that you need to set for your project:

- `DJANGO_SECRET_KEY`: The Django secret key is used as part of the seed for Django's cryptography
- `DJANGO_DB_PASSWORD`: This is the password for the Django's MyMDB database user
- `AWS_ACCESS_KEY_ID`: Your AWS access key
- `AWS_SECRET_ACCESS_KEY_ID`: Your AWS secret access key
- `DJANGO_UPLOAD_S3_BUCKET`: Your bucket name
- `POSTGRES_PASSWORD`: The password for the Postgres database super user (different from the MyMDB database user)
- `DJANGO_ALLOWED_HOSTS`: The domain we'll be serving from (we'll fill this in once we start an EC2 instance)

Next, we define how our containers work together in `docker-compose.yml`:

```
version: '3'

services:
  db:
    build: docker/psql
    restart: always
    ports:
      - "5432:5432"
    environment:
      - DJANGO_DB_USER
      - DJANGO_DB_NAME
      - DJANGO_DB_PASSWORD
  web:
    build: .
    restart: always
    ports:
      - "80:80"
    depends_on:
      - db
    environment:
      - DJANGO_SETTINGS_MODULE
      - DJANGO_SECRET_KEY
      - DJANGO_LOG_LEVEL
      - DJANGO_LOG_FILE
      - DJANGO_ALLOWED_HOSTS
      - DJANGO_DB_NAME
      - DJANGO_DB_USER
```

```
      - DJANGO_DB_PASSWORD
      - DJANGO_DB_HOST
      - DJANGO_DB_PORT
      - DJANGO_CACHE_TIMEOUT
      - AWS_ACCESS_KEY_ID
      - AWS_SECRET_ACCESS_KEY_ID
      - DJANGO_UPLOAD_S3_BUCKET
```

This Compose file describes the two services that make up MyMDB (`db` and `web`). Let's review the configuration options we used:

- `build`: Path to a build context. A build context is, generally speaking, a directory with a `Dockerfile`. So, `db` uses the `psql` directory and `web` uses the `.` directory (the project root directory, which has a `Dockerfile`).
- `ports`: A list of port mappings, describing how to route connections from ports on the host to ports on the container. In our case, we're not changing any ports.
- `environment`: Environment variables for each service. The format we're using implies we're getting the values from our `.env` file. However, you could hardcode values using the `MYVAR=123` syntax.
- `restart`: This is the restart policy for the container. `always` indicates that Docker should always try to restart the container if it stops for any reason.
- `depends_on`: This tells Docker to start the `db` container before the `web` container. However, we still can't be sure that Postgres will manage to start before uWSGI, so we need to check the database is up in our runit script.

Tracing environment variables

Our production configuration relies heavily on environment variables. Let's review the steps we must follow before it can be accessed in Django by `os.getenv()`:

1. List the variable in `.env`
2. Include the variable under the environment option `environment` in `docker-compose.yml`
3. Include the uWSGI ini file variable with `env`
4. Access the variable with `os.getenv`

Running Docker Compose locally

Now that we have configured our Docker containers and Docker Compose, we can run the containers. One of the advantages of Docker Compose is that it can provide the same environment everywhere. This means that we can run Docker Compose locally and get the exact same environment that we'll get in production. There's no need to worry that there's an extra process or a different distribution across environments. Let's run Docker Compose locally.

Installing Docker

To follow along with the rest of this chapter, you must install Docker on your machine. Docker, Inc. provides Docker Community Edition for free from its website: `https://docker.com`. The Docker Community Edition installer is an easy-to-use wizard on Windows and Mac. Docker, Inc. also offers official packages for most major Linux distributions.

Once you have it installed, you'll be able to follow all of the next steps.

Using Docker Compose

To start our containers locally, run the following command:

```
$ docker-compose up -d
```

`docker-compose up` builds and then starts our containers. The `-d` option detaches Compose from our shell.

To check whether our containers are running, we can use `docker ps`:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                          NAMES
0bd7f7203ea0        mymdb_web           "/sbin/my_init"          52 seconds
ago      Up 51 seconds       0.0.0.0:80->80/tcp, 8031/tcp   mymdb_web_1
3b9ecdcf1031        mymdb_db            "docker-entrypoint..."   46 hours
ago        Up 52 seconds       0.0.0.0:5432->5432/tcp         mymdb_db_1
```

To check the Docker logs, you can use the `docker logs` command to note the output of startup scripts:

```
$ docker logs mymdb_web_1
```

To access a shell inside the container (so that you can examine files or view application logs), use this `docker exec` command to start bash:

```
$ docker exec -it mymdb_web_1 bash -l
```

To stop the containers, use the following command:

```
$ docker-compose stop
```

To stop the containers and *delete* them, use the following command:

```
$ docker-compose down
```

When you delete a container, you delete all the data in it. That's not a problem for the Django container as it holds no data. However, if you delete the db container, you *lose the database's data*. Be careful in production.

Sharing your container via a container registry

Now that we have a working container, we may want to make it more widely accessible. Docker has the concept of a container registry. You can push your container to a container registry to make it available either publicly or to just your team.

The most popular Docker container registry is the Docker Hub (`https://hub.docker.com`). You can create an account for free and, at the time of writing this book, each account comes with one free private repository and unlimited public repositories. Most cloud providers also have a docker repository hosting facilities as well (though prices may vary).

The rest of this section assumes that you have a host configured. We'll use Docker Hub as our example, but all the steps are the same regardless of who hosts your container repository.

To share your container, you'll need to do the following things:

1. Log in to a Docker registry
2. Tag our container
3. Push to a Docker registry

Let's start by logging in to a Docker registry:

```
$ docker login -u USERNAME -p PASSWORD docker.io
```

The `USERNAME` and `PASSWORD` values need to be the same as you used for your account on Docker Hub. `docker.io` is the domain of Docker Hub's container registry. If you're using a different container registry host, then you need to change the domain.

Now that we're logged in, let's rebuild and tag our container:

```
$ docker build . -t USERNAME/REPOSITORY:latest
```

Where your `USERNAME` and `REPOSITORY` values are replaced with your values. The `:latest` suffix is the tag for the build. We could have many different tags in the same repository (for example, `development`, `stable`, and `1.x`). Tags in Docker are much like tags in version control; they help us find a particular item quickly and easily. `:latest` is the common tag given to the latest build (though it may not be stable).

Finally, let's push our tagged build to our repository:

```
$ docker push USERNAME/REPOSITORY:latest
```

Docker will show us its progress uploading and then show a SHA256 digest upon success.

When we push a Docker image to a remote repository we need to be mindful of any private data stored on the image. All the files we created or added in `Dockerfile` are contained in the pushed image. Just like we don't want to hard code passwords in code that is stored in a remote repository, we also don't want to store sensitive data (like passwords) in Docker images that might be stored on remote servers. This is another reason we emphasize storing passwords in environment variables rather than hard coding them.

Great! Now you can share the repo with other team members to run your Docker container.

Next, let's launch our container.

Launching containers on a Linux server in the cloud

Now that we have everything working, we can deploy it to the internet. We can use Docker to deploy our containers to any Linux server. Most people who use Docker are using a cloud provider to provide a Linux server host. In our case, we will use AWS.

In the preceding section, when we used `docker-compose`, we were actually using it to send commands to a Docker service running on our machine. Docker Machine provides a way to manage remote servers running Docker. We will use `docker-machine` to start an EC2 instance, which will host our Docker containers.

Starting an EC2 instance can cost money. We'll use an instance that is eligible for the AWS free tier `t2.micro` at the time of writing this book. However, you are responsible for checking the terms of the AWS free tier.

Starting the Docker EC2 VM

We will launch our EC2 VM (called an EC2 instance) into our account's **Virtual Private Cloud** (**VPC**). However, each account has a unique VPC ID. To get your VPC ID, run the following command:

```
$ export AWS_ACCESS_KEY=#your value
$ export AWS_SECRET_ACCESS_KEY=#yourvalue
$ export AWS_DEFAULT_REGION=us-west-2
$ aws ec2 describe-vpcs | grep VpcId
            "VpcId": "vpc-a1b2c3d4",
```

The value used in the preceding code is not a real value.

Now that we know our VPC ID, we can use `docker-machine` to launch an EC2 instance:

```
$ docker-machine create \
     --driver amazonec2 \
     --amazonec2-instance-type t2.micro \
     --amazonec2-vpc-id vpc-a1b2c3d4 \
     --amazonec2-region us-west-2 \
     mymdb-host
```

This tells Docker Machine to launch an EC2 `t2.micro` instance in the `us-west-2` region and the provided VPC. Docker Machine takes care of ensuring that a Docker daemon is installed and started on the server. When referencing this EC2 instance in Docker Machine, we refer to it by the name `mymdb-host`.

When the instance is started, we can ask AWS for the public DNS name for our instance:

```
$ aws ec2 describe-instances | grep -i publicDnsName
```

The preceding command may return multiple copies of the same value even if only one instance is up. Put the result in the `.env` file as `DJANGO_ALLOWED_HOSTS`.

All EC2 instances are protected by a firewall determined by their security group. Docker Machine automatically created a security group for our server when it started our instance. In order for our HTTP requests to make it to our machine, we will need to open port `80` in the `docker-machine` security group, as follows:

```
$ aws ec2 authorize-security-group-ingress \
    --group-name docker-machine \
    --protocol tcp \
    --port 80 \
    --cidr 0.0.0.0/0
```

Now that everything is set up, we can configure `docker-compose` to talk to our remote server and bring up our containers:

```
$ eval $(docker-machine env mymdb-host)
$ docker-compose up -d
```

Congratulations! MyMDB is up in a production environment. Check it out by navigating to the address used in `DJANGO_ALLOWED_HOSTS`.

The instructions here are focused on starting an AWS Linux server. However, all the Docker commands have equivalent options for Google Cloud, Azure, and other major cloud providers. There's even a *generic* option that is made to work with any Linux server, though your mileage may vary depending on the Linux distribution and Docker version.

Shutting down the Docker EC2 VM

Docker machine can also be used to stop VM running Docker as shown in the following snippet:

```
$ export AWS_ACCESS_KEY=#your value
$ export AWS_SECRET_ACCESS_KEY=#yourvalue
$ export AWS_DEFAULT_REGION=us-west-2
$ eval $(docker-machine env mymdb-host)
$ docker-machine stop mymdb-host
```

This will stop the EC2 instance and destroy all the containers in it. If you wish to preserve your DB, ensure that you back up your database by running the preceding `eval` command and then opening a shell using `docker exec -it mymdb_db_1 bash -l`.

Summary

In this chapter, we've launched MyMDB into a production Docker environment on the internet. We've created a Docker container for MyMDB using a Dockerfile. We used Docker Compose to make MyMDB work with a PostgreSQL database (also in a Docker container). Finally, we launched the containers on the AWS cloud using Docker Machine.

Congratulations! You now have MyMDB running.

In the next chapter, we'll make our implementation of Stack Overflow.

6
Starting Answerly

The second project that we will build is a Stack Overflow clone called Answerly. Users who register for Answerly will be able to ask and answer questions. A question's asker will also be able to accept answers to mark them as useful.

In this chapter, we'll do the following things:

- Create our new Django project—Answerly, a Stack Overflow clone
- Create the models for Answerly (`Question` and `Answer`)
- Let users register
- Create forms, views, and templates to let users interact with our models
- Run our code

The code for this project is available online at `https://github.com/tomaratyn/Answerly`.

This chapter won't go deeply into topics already covered in `Chapter 1`, *Building MyMDB*, although it will touch upon many of the same points. Instead, this chapter will focus on going a bit further and introducing new views and third-party libraries.

Let's start our project!

Creating the Answerly Django project

First, let's make a directory for our project:

```
$ mkdir answerly
$ cd answerly
```

All our future commands and paths will be relative to this project directory. A Django project is composed of multiple Django apps.

We'll install Django using `pip`, Python's preferred package manager. We will also track the packages that we install in a `requirements.txt` file:

```
django<2.1
psycopg2<2.8
```

Now, let's install the packages:

```
$ pip install -r requirements.txt
```

Next, let's generate the actual Django project using `django-admin`:

```
$ django-admin startproject config
$ mv config django
```

By default, Django creates a project that will use SQLite, but that's not usable for production; so, we'll follow the best practice of using the same database in development as in production.

Let's open up `django/config/settings.py` and update it to use our Postgres server. Find the line in `settings.py` that starts with `DATABASES`; to use Postgres, change the `DATABASES` value to the following code:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'answerly',
        'USER': 'answerly',
        'PASSWORD': 'development',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

Now that we have our project started and configured, we can create and install the two Django apps we'll make as part of this project:

```
$ cd django
$ python manage.py startapp user
$ python manage.py startapp qanda
```

A Django project is composed of apps. Django apps are where all the functionalities and code live. Models, forms, and templates all belong to Django apps. An app, like every other Python module, should have a clearly defined scope. In our case, we have two apps each with different roles. The `qanda` app will be responsible for the question and answer functionality of our app. The `user` app will be responsible for user management of our app. Each of them will also rely on other apps and Django's core functionality to work effectively.

Now, let's install our apps in our project by updating `django/config/settings.py`:

```
INSTALLED_APPS = [
    'user',
    'qanda',

    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Now that Django knows about our app, let's install start with the models for `qanda`.

Creating the Answerly models

Django is particularly helpful for creating data-driven apps. Models, representing the data in the apps, are often the core of these apps. Django encourages this with the best practice of *fat models, thin views, dumb templates*. The advice encourages us to place business logic in our models rather than our views.

Let's start building our `qanda` models with the `Question` model.

Creating the Question model

We'll create our `Question` model in `django/qanda/models.py`:

```
from django.conf import settings
from django.db import models
from django.urls.base import reverse


class Question(models.Model):
    title = models.CharField(max_length=140)
    question = models.TextField()
    user = models.ForeignKey(to=settings.AUTH_USER_MODEL,
                             on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('questions:question_detail', kwargs={'pk': self.id})

    def can_accept_answers(self, user):
        return user == self.user
```

A `Question` model, like all Django models, is derived from `django.db.models.Model`. It has the following four fields that will become columns in a `questions_question` table:

- `title`: A character field that will become a `VARCHAR` column of maximum 140 characters.
- `question`: This is the body of the question. Since we can't predict how long this will be, we use a `TextField`, which will become a `TEXT` column.The `TEXT` columns don't have a size limit.
- `user`: This will create a foreign key to the project's configured user model. In our case, we will go with the default `django.contrib.auth.models.User` that comes with Django. However, it's still recommended to not hardcode this when we can avoid it.
- `created`: This will be automatically set to the date and time that the `Question` model was created.

`Question` also implements the following two methods commonly seen on Django models (`__str__` and `get_absolute_url`):

- `__str__()`: This tells Python how to convert our model to a string. This is useful in the admin backend, our own templates, and in debugging.
- `get_absolute_url()`: This is a commonly implemented method that lets the model return the path of a URL to view this model. Not all models need this method. Django's built-in views, such as `CreateView`, will use this method to redirect the user to the view after the model is created.

Finally, in the spirit of *fat models*, we also have `can_accept_answers()`. The decision of who can accept an `Answer` to a `Question` lies with the `Question`. Currently, only the user who asked the question can accept an answer.

Now that we have the `Question` s, we naturally need `Answer` s.

Creating the Answer model

We'll create the `Answer` model in the `django/questions/models.py` file as shown in the following code:

```
from django.conf import settings
from django.db import models

class Question(model.Models):
    # skipped

class Answer(models.Model):
    answer = models.TextField()
    user = models.ForeignKey(to=settings.AUTH_USER_MODEL,
                             on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True)
    question = models.ForeignKey(to=Question,
                                 on_delete=models.CASCADE)
    accepted = models.BooleanField(default=False)

    class Meta:
        ordering = ('-created', )
```

The `Answer` model has five fields and a `Meta` class. Let's take a look at the fields first:

- `answer`: This is an unlimited text field for the user's answer. `answer` will become a `TEXT` column.
- `user`: This will create a foreign key to the user model that our project has been configured to use. The user model will gain a new `RelatedManager` under the name `answer_set`, which will be able to query all the `Answer` s for a user.
- `question`: This will create a foreign key to our `Question` model. `Question` will also gain a new `RelatedManager` under the name `answer_set`, which will be able to query all the `Answer` s to a `Question`.
- `created`: This will be set to the date and time when the `Answer` was created.
- `accepted`: This is a Boolean that will be set to `False` by default. We'll use it to mark accepted answers.

A model's `Meta` class lets us set metadata for our model and table. For `Answer`, we're using the `ordering` option to ensure that all queries will be ordered by `created`, in descending order. In this way, we ensure that the newest answers will be listed first, by default.

Now that we have `Question` and `Answer` models, we will need to create migrations to create their tables in the database.

Creating migrations

Django comes with a built-in migration library. This is part of Django's *batteries included* philosophy. Migrations provide a way to manage the changes that we will need to make to our schema. Whenever we make a change to a model, we can use Django to generate a migration, which will contain the instructions on how to create or change the schema to fit the new model's definition. To make the change to our database, we will apply the schema.

Like many operations we perform on our project, we'll use the `manage.py` script Django provides for our project:

```
$ python manage.py makemigrations
 Migrations for 'qanda':
  qanda/migrations/0001_initial.py
    - Create model Answer
    - Create model Question
    - Add field question to answer
    - Add field user to answer
$ python manage.py migrate
Operations to perform:
```

```
  Apply all migrations: admin, auth, contenttypes, qanda, sessions
Running migrations:
  Applying qanda.0001_initial... OK
```

Now that we've created the migrations and applied them, let's set up a base template for our project so that our code works well.

Adding a base template

Before we create our views, let's create a base template. Django's template language allows templates to inherit from each other. A base template is a template that all our other project's templates will extend. This will give our entire project a common look and feel.

Since a project is composed of multiple apps and they will all use the same base template, a base template belongs to the project, not to any particular app. This is a rare exception to the rule that everything lives in an app.

To add a project-wide templates directory, update `django/config/settings.py`. Check the `TEMPLATES` setting and update it to this:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'templates')
        ],
        'APP_DIRS': True,
        'OPTIONS': {
                # skipping rest of options.
        },
    },
]
```

In particular, the `DIRS` option for the `django.template.backends.django.DjangoTemplates` setting sets a project-wide template directory that will be searched. `'APP_DIRS': True` means that each installed app's `templates` directory will also be searched. In order for Django to search `django/templates`, we must add `os.path.join(BASE_DIR, 'templates')` to the `DIRS` list.

Creating base.html

Django comes with its own template language eponymously called the Django Template Language. Django templates are text files, which are rendered using a dictionary (called a context) to look up values. A template can also include tags (which use the `{% tag argument %}` syntax). A template can print values from its context using the `{{ variableName }}` syntax. Values can be sent to filters to tweak them before being displayed (for example, `{{ user.username | uppercase }}` will print the user's username with all uppercase characters). Finally, the `{# ignored #}` syntax can comment out multiple lines of text.

We'll create our base template in `django/templates/base.html`:

```
{% load static %}
<!DOCTYPE html>
<html lang="en" >
<head >
  <meta charset="UTF-8" >
  <title >{% block title %}Answerly{% endblock %}</title >
  <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.
min.css"
      rel="stylesheet">
  <link
href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.m
in.css"
      rel="stylesheet">
  <link rel="stylesheet" href="{% static "base.css" %}" >
</head >
<body >
<nav class="navbar navbar-expand-lg  bg-light" >
  <div class="container" >
    <a class="navbar-brand" href="/" >Answerly</a >
    <ul class="navbar-nav" >
    </ul >
  </div >
</nav >
<div class="container" >
  {% block body %}{% endblock %}
</div >
</body >
</html >
```

We won't go over this HTML, but it's worth reviewing the Django template tags involved:

- `{% load static %}`: `load` lets us load template tag libraries that aren't available by default. In this case, we're loading the static library, which provides the `static` tag. The library and tag don't always share their name. This is provided with Django by the `django.contrib.static` app.
- `{% block title %}Answerly{% endblock %}`: Blocks let us define areas that templates can override when extending this template.
- `{% static 'base.css' %}`: The `static` tag (loaded in from the preceding `static` library) uses the `STATIC_URL` setting to create a reference to a static file. In this case, it will return `/static/base.css`. As long as the file is in a directory listed in `settings.STATICFILES_DIRS` and Django is in debug mode, Django will serve that file for us. For production, refer to `Chapter 9`, *Deploying Answerly*.

That's enough for our `base.html` file to start. We'll update the navigation in `base.html` later, in the *Updating base.html navigation* section.

Next, let's configure Django to know how to find our `base.css` file by configuring static files.

Configuring static files

Next, let's configure a directory for project-wide static files in `django/config/settings.py`:

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

This will tell Django that any file in `django/static/` should be served while Django is in debug mode. For production, refer to `Chapter 9`, *Deploying Answerly*.

Let's put some basic CSS in `django/static/base.css`:

```
nav.navbar {
  margin-bottom: 1em;
}
```

Now that we have created the foundation, let's create `AskQuestionView`.

Letting users post questions

We will now create a view for letting users post questions that they need answered.

Django follows **Model-View-Template** (**MVT**) pattern separate model, control, and presentation logic and encourage reusability. Models represent the data we'll store in the database. Views are responsible for handling a request and returning a response. Views should not have HTML. Templates are responsible for the body of a response and defining the HTML. This separation of responsibilities has proven to make it easy to write code.

To let users post questions, we'll perform the following steps:

1. Make a form to process the questions
2. Make a view that uses Django forms to create questions
3. Make a template that renders the form in HTML
4. Add a `path` to the view

First, let's make the `QuestionForm` class.

Ask question form

Django forms serve two purposes. They make it easy to render the body of a form to receive user input. They also validate the user input. When a form is instantiated, it can be given initial values (by the `intial` parameter) and data to validate (by the `data` parameter). A form which has been provided data is said to be bound.

Much of the power of Django comes from how easy it is to join models, forms, and views together to build features.

We'll make our form in `django/qanda/forms.py`:

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

from qanda.models import Question


class QuestionForm(forms.ModelForm):
    user = forms.ModelChoiceField(
```

```
        widget=forms.HiddenInput,
        queryset=get_user_model().objects.all(),
        disabled=True,
    )

    class Meta:
        model = Question
        fields = ['title', 'question', 'user', ]
```

`ModelForm` makes creating forms from Django models easier. We use the inner `Meta` class of `QuestionForm` to specify the model and fields that are part of the form.

By adding a `user` field, we're able to override how Django renders the `user` field. We tell Django to use the `HiddenInput` widget, which will render the field as `<input type='hidden'>`. The `queryset` argument lets us restrict the users that are valid values (in our case, all users are valid). Finally, the `disabled` argument says that we will ignore any values provided by `data` (that is, from a request) and rely on the `initial` values we provide to the form.

Now that we know how to render and validate a question form, let's create our view.

Creating AskQuestionView

We will create our `AskQuestionView` class in `django/qanda/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView

from qanda.forms import QuestionForm
from qanda.models import Question

class AskQuestionView(LoginRequiredMixin, CreateView):
    form_class = QuestionForm
    template_name = 'qanda/ask.html'

    def get_initial(self):
        return {
            'user': self.request.user.id
        }

    def form_valid(self, form):
        action = self.request.POST.get('action')
        if action == 'SAVE':
            # save and redirect as usual.
```

```
            return super().form_valid(form)
        elif action == 'PREVIEW':
            preview = Question(
                question=form.cleaned_data['question'],
                title=form.cleaned_data['title'])
            ctx = self.get_context_data(preview=preview)
            return self.render_to_response(context=ctx)
        return HttpResponseBadRequest()
```

`AskQuestionView` is derived from `CreateView` and uses the `LoginRequiredMixin`. The `LoginRequiredMixin` ensures that any request made by a user who is not logged in will be redirected to the login page. The `CreateView` knows to render the template for `GET` requests and to validate the form on `POST` requests. If a form is valid, `CreateView` will call `form_valid`. If the form is not valid, `CreateView` will re-render the template.

Our `form_valid` method overrides the original `CreateView` method to support a save and preview mode. When we want to save, we will call the original `form_valid` method. The original method saves the new question and returns an HTTP response that redirects the user to the new question (using `Question.get_absolute_url()`). When we want to preview the question, we will re-render our template with the new `preview` variable in our template's context.

When our view is instantiating the form, it will pass the result of `get_initial()` as the `initial` argument and the `POST` data as the `data` argument.

Now that we have our view, let's create `ask.html`.

Creating ask.html

Let's create our template in `django/qanda/ask.html`:

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

{% load markdownify %}
{% load crispy_forms_tags %}

{% block title %} Ask a question {% endblock %}

{% block body %}
  <div class="col-md-12" >
    <h1 >Ask a question</h1 >
    {% if preview %}
      <div class="card question-preview" >
```

```
          <div class="card-header" >
            Question Preview
          </div >
          <div class="card-body" >
            <h1 class="card-title" >{{ preview.title }}</h1>
            {{ preview.question |  markdownify }}
          </div >
        </div >
      {% endif %}

      <form method="post" >
        {{ form | crispy }}
        {% csrf_token %}
        <button class="btn btn-primary" type="submit" name="action"
                value="PREVIEW" >
          Preview
        </button >
        <button class="btn btn-primary" type="submit" name="action"
                value="SAVE" >
          Ask!
        </button >
      </form >
    </div >
  {% endblock %}
```

This template uses our `base.html` template and puts all its HTML in the `blocks` defined by there. When we render the template, Django renders `base.html` and then fills in the values of the blocks with the contents defined in `ask.html`.

`ask.html` also loads two third-party tag libraries, `markdownify` and `crispy_forms_tags`. `markdownify` provides the `markdownify` filter used in the preview card's body (`{{preview.question | markdownify}}`). The `crispy_forms_tags` library provides the `crispy` filter, which applies Bootstrap 4 CSS classes to help the Django form render nicely.

Each of these libraries needs to be installed and configured, which we do in the following sections (*Installing and configuring Markdownify* and *Installing and configuring Django Crispy Forms*, respectively).

The following are a few more new tags that `ask.html` shows us:

- `{% if preview %}`: This demonstrates how to use an `if` statement in the Django template language. We only want to render a preview of the `Question` if we have a `preview` variable in our context.

- `{% csrf_token %}`: This tag adds the expected CSRF token to our form. CSRF tokens help protect us against malicious scripts trying to submit data on behalf of an innocent but logged-in user; refer to `Chapter 3`, *Posters, Headshots, and Security*, for more information. In Django, CSRF tokens are not optional, and `POST` requests missing a CSRF token will not be processed.

Let's take a closer look at those third-party libraries, starting with Markdownify.

Installing and configuring Markdownify

Markdownify is a Django app available on the **Python Package Index** (**PyPI**) created by R Moelker and Erwin Matijsen and licensed under the MIT license (a popular open source license). Markdownify provides the Django template filter `markdownify`, which will convert Markdown to HTML.

Markdownify works by using the **python-markdown** package to convert Markdown to HTML. Marodwnify then uses Mozilla's `bleach` library to sanitize the resultiant HTML from Cross Site Scripting (**XSS**) attacks. The result is then returned to the template for output.

To install Markdownify, let's add it to our `requirements.txt` file:

```
django-markdownify==0.2.2
```

Then, run `pip` to install it:

```
$ pip install -r requirements.txt
```

Now, we will need to add `markdownify` to our list of `INSTALLED_APPS` in `django/config/settings.py`.

The last step is to configure Markdownify to let it know which HTML tags to whitelist. Add the following settings to `settings.py`:

```
MARKDOWNIFY_STRIP = False
MARKDOWNIFY_WHITELIST_TAGS = [
    'a', 'blockquote', 'code', 'em', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6',
    'h7', 'li', 'ol', 'p', 'strong', 'ul',
]
```

This will whitiest all the text, list, and heading tags we want available to our users. Setting `MARKDOWNIFY_STRIP` to `False` tells Markdownify to HTML encode (rather than strip) any other HTML tag.

Now that we've configured Markdownify, let's install and configure Django Crispy Forms.

Installing and configuring Django Crispy Forms

Django Crispy Forms is a third-party Django app available on PyPI. Miguel Araujo is the development lead. It is licensed under the MIT license. Django Crispy Forms is one of the most popular Django libraries because it makes it so easy to render pretty (crisp) forms.

One of the problems we encounter in Django is that when Django renders a field it will render it something like this:

```
<label for="id_title">Title:</label>
<input
      type="text" name="title" maxlength="140" required id="id_title" />
```

However, in order to style that form nicely, for example, using Bootstrap 4, we would like to render something more like this:

```
<div class="form-group">
<label for="id_title" class="form-control-label  requiredField">
   Title
</label>
<input type="text" name="title" maxlength="140"
  class="textinput textInput form-control" required="" id="id_title">
</div>
```

Sadly, Django doesn't provide hooks that would let us easily wrap the field in a `div` with class `form-group`, or add CSS classes such as `form-control` or `form-control-label`.

Django Crispy Forms solves this with its `crispy` filter. If we send a form into it by performing `{{ form | crispy}}`, Django Crispy Forms will correctly transform the form's HTML and CSS to work with a variety of CSS frameworks (including Zurb Foundation, Bootstrap 3, and Bootstrap 4). You can further customize the form's rendering through more advanced usage of Django Crispy Forms, but we won't be doing that in this chapter.

To install Django Crispy Forms, let's add it to our `requirements.txt` and install it using `pip`:

```
$ echo "django-crispy-forms==1.7.0" >> requirements.txt
$ pip install -r requirements.txt
```

Now, we will need to install it as a Django app in our project by editing `django/config/settings.py` and adding `'crispy_forms'` to our list of `INSTALLED_APPS`.

Next, we will need to configure our project so that Django Crispy Forms knows to use the Bootstrap 4 template pack. Update `django/config/settings.py` with a new config:

```
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

Now that we've installed all the libraries our template relies on, we can configure Django to route requests to our `AskQuestionView`.

Routing requests to AskQuestionView

Django routes requests using a URLConf. It's a list of `path()` objects that a request's path is matched against. The view of the first matching `path()` gets to process the request. A URLConf can include another URLConf. A project's settings defines its root URLConf (in our case, `django/config/urls.py`).

Defining all the `path()` objects for all the views in a project in the root URLConf can get messy and makes the apps less reusable. It's often convenient to put a URLConf (usually in a `urls.py` file) in each app. Then, the root URLConf can use the `include()` function to include other apps' URLConfs to route requests.

Let's create a URLConf for our `qanda` app in `django/qanda/urls.py`:

```
from django.urls.conf import path

from qanda import views

app_name = 'qanda'
urlpatterns = [
    path('ask', views.AskQuestionView.as_view(), name='ask'),
]
```

A path has at least two components:

- First, a string defining the matching path. This may have named parameters that will be passed to the view. We'll see an example of this later, in the *Routing requests to the QuestionDetail view* section.

- Second, a callable that takes a request and returns a response. If your view is a function (also known as a **Function-Based View** (**FBV**)), then you can just pass a reference to your function. If you're using a **Class-Based View** (**CBV**), then you can use its `as_view()` class method to return the required callable.
- Optionally, a `name` parameter which we can use to reference this `path()` object in our view or template (for example, like the `Question` model does in its `get_absolute_url()` method).

It is very strongly recommended that you name all your `path()` objects.

Now, let's update our root URLConf to include the `qanda` URLConf:

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

import qanda.urls

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

This means that requests to `answerly.example.com/ask` will route to our `AskQuestionView`.

A quick review of the section

In this section, we have performed the following actions:

- Created our first form, `QuestionForm`
- Created `AskQuestionView` that uses the `QuestionForm` to create `Question` s
- Created a template to render `AskQuestionView` and `QuestionForm`
- Installed and configured third-party libraries that provide filters for our template

Now, let's allow our users to view questions with a `QuestionDetailView` class.

Creating QuestionDetailView

The `QuestionDetailView` has to offer quite a bit of functionality. It must be able to do the following things:

- Show the question
- Show all the answers
- Let users post additional answers
- Let the asker accept answer(s)
- Let the asker reject previously-accepted answers

Although `QuestionDetailView` won't process any forms, it will have to display many forms, leading to a complicated template. This complexity will give us a chance to note how to split a template up into separate subtemplates to make our code more readable.

Creating Answer forms

We'll need to make two forms to make `QuestionDetailView` work as described in the preceding section:

- `AnswerForm`: For users to post their answers
- `AnswerAcceptanceForm`: For the question's asker to accept or reject answers

Creating AnswerForm

The `AnswerForm` will have to reference a `Question` model instance and a user because both are required to create an `Answer` model instance.

Let's add our `AnswerForm` to `django/qanda/forms.py`:

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

from qanda.models import Answers

class AnswerForm(forms.ModelForm):
    user = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=get_user_model().objects.all(),
        disabled=True,
    )
```

```
    question = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=Question.objects.all(),
        disabled=True,
    )

    class Meta:
        model = Answer
        fields = ['answer', 'user', 'question', ]
```

The `AnswerForm` class looks a lot like the `QuestionForm` class, though with slightly differently named fields. It uses the same technique of preventing a user from tampering with the `Question` associated with an `Answer` just as `QuestionForm` used to prevent tampering with the user of `Question`.

Next, we'll create a form to accept an `Answer`.

Creating AnswerAcceptanceForm

An `Answer` is accepted if its `accepted` field is `True`. We'll use a simple form to edit this field:

```
class AnswerAcceptanceForm(forms.ModelForm):
    accepted = forms.BooleanField(
        widget=forms.HiddenInput,
        required=False,
    )

    class Meta:
        model = Answer
        fields = ['accepted', ]
```

Using `BooleanField` comes with a small wrinkle. If we want `BooleanField` to accept `False` values as well as `True` values, we must set `required=False`. Otherwise, `BooleanField` will get confused when it gets a `False` value, thinking that it actually didn't receive a value.

We use a hidden input because we don't want users checking a checkbox and then having to click on submit. Instead, for each answer, we'll generate an accept form and a reject form, which the user can just submit with one click.

Next, let's write the `QuestionDetailView` class.

Creating QuestionDetailView

Now that we have the forms we'll use, we can create `QuestionDetailView` in `django/qanda/views.py`:

```
from django.views.generic import DetailView

from qanda.forms import AnswerForm, AnswerAcceptanceForm
from qanda.models import Question


class QuestionDetailView(DetailView):
    model = Question

    ACCEPT_FORM = AnswerAcceptanceForm(initial={'accepted': True})
    REJECT_FORM = AnswerAcceptanceForm(initial={'accepted': False})

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        ctx.update({
            'answer_form': AnswerForm(initial={
                'user': self.request.user.id,
                'question': self.object.id,
            })
        })
        if self.object.can_accept_answers(self.request.user):
            ctx.update({
                'accept_form': self.ACCEPT_FORM,
                'reject_form': self.REJECT_FORM,
            })
        return ctx
```

`QuestionDetailView` lets Django's `DetailView` do most of the work. `DetailView` gets a `Question` `QuerySet` out of the default manager of `Question` (`Question.objects`). `DetailView` then uses the `QuerySet` to get a `Question` based on the `pk` it received in the path of the URL. `DetailView` also knows which template to render based on our app and model name (`appname/modelname_detail.html`).

The only area where we've had to customize behavior of `DetailView` is `get_context_data()`. `get_context_data()` provides the context used to render the template. In our case, we use the method to add the forms we want rendered to context.

Next, let's make the template for `QuestionDetailView`.

Creating question_detail.html

Our template for the `QuestionDetailView` will work slightly differently to our previous templates.

Here's what we'll put in `django/qanda/templates/qanda/question_detail.html`:

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

{% block title %}{{ question.title }} - {{ block.super }}{% endblock %}

{% block body %}
  {% include "qanda/common/display_question.html" %}
  {% include "qanda/common/list_answers.html" %}
  {% if user.is_authenticated %}
    {% include "qanda/common/question_post_answer.html" %}
  {% else %}
    <div >Login to post answers.</div >
  {% endif %}
{% endblock %}
```

The preceding template seemingly doesn't do anything itself. Instead, we use the `{% include %}` tag to include other templates inside this template, to make organizing our code simpler. `{% include %}` passes the current context to the new template, renders it, and inserts it in place.

Let's take a look at each of these sub templates in turn, staring with `dispaly_question.html`.

Creating the display_question.html common template

We've put the HTML to display a question into its own sub template. This template can then be included by other templates to render a `question`.

Let's create it in `django/qanda/templates/qanda/common/display_question.html`:

```
{% load markdownify %}
<div class="question" >
  <div class="meta col-sm-12" >
    <h1 >{{ question.title }}</h1 >
    Asked by {{ question.user }} on {{ question.created }}
  </div >
  <div class="body col-sm-12" >
    {{ question.question|markdownify }}
  </div >
</div >
```

The HTML itself is pretty simple, and there are no new tags here. We reuse the `markdownify` tag and library that we have previously configured.

Next, let's look at the answer list template.

Creating list_answers.html

The answer list template has to list all the answers for the question and also render whether the answer is accepted. If the user can accept (or reject) answers, then those forms are rendered too.

Let's create the template in `django/qanda/templates/qanda/view_questions/question_answers.html`:

```
{% load markdownify %}
<h3 >Answers</h3 >
<ul class="list-unstyled answers" >
  {% for answer in question.answer_set.all %}
    <li class="answer row" >
      <div class="col-sm-3 col-md-2 text-center" >
        {% if answer.accepted %}
          <span class="badge badge-pill badge-success" >Accepted</span >
        {% endif %}
        {% if answer.accepted and reject_form %}
          <form method="post"
                action="{% url "qanda:update_answer_acceptance"
pk=answer.id %}" >
            {% csrf_token %}
            {{ reject_form }}
            <button type="submit" class="btn btn-link" >
              <i class="fa fa-times" aria-hidden="true" ></i>
              Reject
```

```
            </button >
          </form >
        {% elif accept_form %}
          <form method="post"
                action="{% url "qanda:update_answer_acceptance"
pk=answer.id %}" >
            {% csrf_token %}
            {{ accept_form }}
            <button type="submit" class="btn btn-link" title="Accept
answer" >
              <i class="fa fa-check-circle" aria-hidden="true"></i >
              Accept
            </button >
          </form >
        {% endif %}
      </div >
      <div class="col-sm-9 col-md-10" >
        <div class="body" >{{ answer.answer|markdownify }}</div >
        <div class="meta font-weight-light" >
          Answered by {{ answer.user }} on {{ answer.created }}
        </div >
      </div >
    </li >
  {% empty %}
    <li class="answer" >No answers yet!</li >
  {% endfor %}
</ul >
```

Two things to observe about this template are as follows:

- There's a rare bit of logic in the template, `{% if answer.accepted and reject_form %}`. Generally, templates should be dumb and avoid knowing about business logic. However, avoiding this would have created a more complex view. This is a trade-off that we must always evaluate on a case-by-case basis.
- The `{% empty %}` tag is related to our `{% for answer in question.answer_set.all %}` loop. The `{% empty %}` is used in the case of an empty list, much like the Python's `for ... else` syntax.

Next, let's take a look at the post answer template.

Creating the post_answer.html template

In the next template we're going to create, the user can post and preview their answer.

Let's create our next template in `django/qanda/templates/qanda/common/post_answer.html`:

```
{% load crispy_forms_tags %}

<div class="col-sm-12" >
  <h3 >Post your answer</h3 >
  <form method="post"
        action="{% url "qanda:answer_question" pk=question.id %}" >
    {{ answer_form | crispy }}
    {% csrf_token %}
    <button class="btn btn-primary" type="submit" name="action"
            value="PREVIEW" >Preview
    </button >
    <button class="btn btn-primary" type="submit" name="action"
            value="SAVE" >Answer
    </button >
  </form >
</div >
```

This template is quite simple, sampling rendering the `answer_form` using the `crispy` filter.

Now that we have all our subtemplates done, let's create a `path` to route requests to `QuestionDetailView`.

Routing requests to the QuestionDetail view

To be able to route requests to our `QuestionDetailView`, we need to add it to the URLConf in `django/qanda/urls.py`:

```
    path('q/<int:pk>', views.QuestionDetailView.as_view(),
         name='question_detail'),
```

In the preceding code, we see `path` taking a named parameter `pk`, which must be an integer. This will be passed to the `QuestionDetailView` and available in the `kwargs` dictionary. `DetailView` will rely on the presence of this argument to know which `Question` to retrieve.

Next, we'll create some of the form-related views we referenced in our templates. Let's start with the `CreateAnswerView` class.

Creating the CreateAnswerView

The `CreateAnswerView` class will be used to create and preview `Answer` model instance for a `Question` model instance.

Let's create it in `django/qanda/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView

from qanda.forms import AnswerForm


class CreateAnswerView(LoginRequiredMixin, CreateView):
    form_class = AnswerForm
    template_name = 'qanda/create_answer.html'

    def get_initial(self):
        return {
            'question': self.get_question().id,
            'user': self.request.user.id,
        }

    def get_context_data(self, **kwargs):
        return super().get_context_data(question=self.get_question(),
                                        **kwargs)

    def get_success_url(self):
        return self.object.question.get_absolute_url()

    def form_valid(self, form):
        action = self.request.POST.get('action')
        if action == 'SAVE':
            # save and redirect as usual.
            return super().form_valid(form)
        elif action == 'PREVIEW':
```

```
            ctx =
self.get_context_data(preview=form.cleaned_data['answer'])
            return self.render_to_response(context=ctx)
        return HttpResponseBadRequest()

    def get_question(self):
        return Question.objects.get(pk=self.kwargs['pk'])
```

The `CreateAnswerView` class follows a similar pattern to the `AskQuestionView` class:

- It's a `CreateView`
- It's protected by `LoginRequiredMixin`
- It uses `get_initial()` to provide initial arguments to its form so malicious users can't tamper with the question or user associated with the answer
- It uses `form_valid()` to perform a preview or save operation

The main difference is that we will need to add a `get_question()` method in `CreateAnswerView` to retrieve the question we're answering. `kwargs['pk']` will be populated by the `path` we'll create (just like we did for `QuestionDetailView`).

Next, let's create the template.

Creating create_answer.html

This template will be able to leverage the common template elements we've already created to make rendering the question and answer forms easier.

Let's create it in `django/qanda/templates/qanda/create_answer.html`:

```
{% extends "base.html" %}
{% load markdownify %}

{% block body %}
  {% include 'qanda/common/display_question.html' %}
  {% if preview %}
    <div class="card question-preview" >
      <div class="card-header" >
        Answer Preview
      </div >
      <div class="card-body" >
        {{ preview|markdownify }}
      </div >
    </div >
  {% endif %}
```

```
    {% include 'qanda/common/post_answer.html' with answer_form=form %}
{% endblock %}
```

The preceding template introduces a new use of `{% include %}`. When we use the `with` argument, we can then pass a series of new names that values should have in the subtemplate's context. In our case, we will only add the `answer_form` to the context of `post_answer.html`. The rest of the context is still passed to `{% include %}`. We can prevent the rest of the context being passed if we add `only` as the last argument to `{% include %}`.

Routing requests to CreateAnswerView

The final step is to connect the `CreateAnswerView` to the `qanda` URLConf by adding a new `path` to the `urlpatterns` list in `qanda/urls.py`:

```
    path('q/<int:pk>/answer', views.CreateAnswerView.as_view(),
         name='answer_question'),
```

Next, we'll make a view to process the `AnswerAcceptanceForm`.

Creating UpdateAnswerAcceptanceView

The `accept_form` and `reject_form` variables we use in the `list_answers.html` template need a view to process their form submissions. Let's add it to `django/qanda/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import UpdateView

from qanda.forms import AnswerAcceptanceForm
from qanda.models import Answer

class UpdateAnswerAcceptance(LoginRequiredMixin, UpdateView):
    form_class = AnswerAcceptanceForm
    queryset = Answer.objects.all()

    def get_success_url(self):
        return self.object.question.get_absolute_url()

    def form_invalid(self, form):
        return HttpResponseRedirect(
            redirect_to=self.object.question.get_absolute_url())
```

`UpdateView` works like a mix of `DetailView` (since it works on a single model) and `CreateView` (since it processes a form). Both `CreateView` and `UpdateView` share a common ancestor: `ModelFormMixin`. `ModelFormMixin` provides us with the hooks we've used so often in the past: `form_valid()`, `get_success_url()`, and `form_invalid()`.

Thanks to the simplicity of this form, we will just respond to an invalid form by redirecting the user to the question.

Next, let's add it to our URLConf in `django/qanda/urls.py` file:

```
    path('a/<int:pk>/accept', views.UpdateAnswerAcceptance.as_view(),
         name='update_answer_acceptance'),
```

Remember to have a parameter named `pk` in your `path()` object's first argument so that `UpdateView` can retrieve the correct `Answer`.

Next, let's create a daily list of questions.

Creating the daily questions page

To help people find questions, we'll create a list of each day's questions.

Django offers views to create yearly, monthly, weekly, and daily archive views. In our case, we'll use the `DailyArchiveView`, but they all work basically the same. They take a date from the URL's path and search for everything related during that period.

Let's build a daily question list using Django's `DailyArchiveView`.

Creating DailyQuestionList view

Let's add our `DailyQuestionList` view to `django/qanda/views.py`:

```
from django.views.generic import DayArchiveView

from qanda.models import Question

class DailyQuestionList(DayArchiveView):
    queryset = Question.objects.all()
    date_field = 'created'
    month_format = '%m'
    allow_empty = True
```

`DailyQuestionList` need not override any methods of `DayArchiveView` just to let Django do the work. Let's look at how it does it.

`DayArchiveView` expects to get a day, month, and year in the URL's path. We can specify the format of these using `day_format`, `month_format`, and `year_format`. In our case, we change the expected format to `'%m'` so that the month is parsed as a number instead of the default `'%b'`, which is the short name of the month. These formats are the same, Python's standard `datetime.datetime.strftime`. Once `DayArchiveView` has the date, it uses that date to filter the provided `queryset` using field named in the `date_field` attribute. The `queryset` is ordered by date. If `allow_empty` is `True`, then results will be rendered, otherwise a 404 exception is thrown, for days with no items to list. To render the template, the object list is passed to the template much like a `ListView`. The default template is assumed to follow the `appname/modelname_archive_day.html` format.

Next, let's create the template for this view.

Creating the daily question list template

Let's add our template to `django/qanda/templates/qanda/question_archive_day.html`:

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

{% block title %} Questions on {{ day }} {% endblock %}

{% block body %}
  <div class="col-sm-12" >
    <h1 >Highest Voted Questions of {{ day }}</h1 >
    <ul >
      {% for question in object_list %}
        <li >
          {{ question.votes }}
          <a href="{{ question.get_absolute_url }}" >
            {{ question }}
          </a >
          by
            {{ question.user }}
          on {{ question.created }}
        </li >
      {% empty %}
        <li>Hmm... Everyone thinks they know everything today.</li>
      {% endfor %}
```

```
    </ul >
    <div>
      {% if previous_day %}
        <a href="{% url "qanda:daily_questions" year=previous_day.year
month=previous_day.month day=previous_day.day %}" >
          << Previous Day
        </a >
      {% endif %}
      {% if next_day %}
        <a href="{% url "qanda:daily_questions" year=next_day.year
month=next_day.month day=next_day.day %}" >
          Next Day >>
        </a >
      {% endif %}
    </div >
  </div >
{% endblock %}
```

The list of questions is much like one would expect, that is, a `<ul>` tag with a `{% for %}` loop creating `<li>` tags with links.

One of the conveniences of the `DailyArchiveView` (and all the date archive views) is that they provide their template's context with next and previous dates. These dates let us create a kind of pagination across dates.

Routing requests to DailyQuestionLists

Finally, we'll create a `path` to our `DailyQuestionList` view so that we can route requests to it:

```
path('daily/<int:year>/<int:month>/<int:day>/',
     views.DailyQuestionList.as_view(),
     name='daily_questions'),
```

Next, let's create a view to represent *today's* questions.

Getting today's question list

Having a daily archive is good, but we want to provide a convenient way to access today's archive. We'll use a `RedirectView` to always redirect the user to the `DailyQuestionList` of today's date.

Let's add it to `django/qanda/views.py`:

```
class TodaysQuestionList(RedirectView):
    def get_redirect_url(self, *args, **kwargs):
        today = timezone.now()
        return reverse(
            'questions:daily_questions',
            kwargs={
                'day': today.day,
                'month': today.month,
                'year': today.year,
            }
        )
```

`RedirectView` is a simple view that returns a 301 or 302 redirect response. We use Django's `django.util.timezone` to get today's date according to how Django has been configured. By default, Django is configured using **Coordinated Universal Time** (**UTC**). Due to the complexity of time zones, it's often simplest to track everything in UTC and then adjust the display on the client side.

We've now created all the views for our initial `qanda` app, letting users ask and answer questions. The asker can also accept answer(s) to their question.

Next, let's actually let the users log in, log out, and register with a `user` app.

Creating the user app

As we mentioned before, a Django app should have a clear scope. To that end, we'll create a separate Django app to manage users, which we will call `user`. We shouldn't place our user management code in `qanda` or the `Question` model in the `user` app.

Let's create the app using `manage.py`:

```
$ python manage.py startapp user
```

Then, add it to our list of `INSTALLED_APPS` in `django/config/settings.py`:

```
INSTALLED_APPS = [
    'user',
    'qanda',

    'markdownify',
    'crispy_forms',
```

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

It's particularly important to keep the `user` app *before* the `admin` app, as they will both define login templates. The app that comes first will have their login template resolved first. We don't want our users redirected to the admin app.

Next, let's create a URLConf for our `user` app in `django/user/urls.py`:

```
from django.urls import path

import user.views

app_name = 'user'
urlpatterns = [
]
```

Now, we'll have the main URLConf in `django/config/urls.py` include the `user` app's URLConf:

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

import qanda.urls
import user.urls

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

Now that we have our app configured, we can add our login and logout views.

Using Django's LoginView and LogoutView

To provide the login and logout functionalities, we'll use views provided by the `django.contrib.auth` app. Let's update the `django/users/urls.py` to reference them:

```
from django.urls import path

import user.views

app_name = 'user'
urlpatterns = [
    path('login', LoginView.as_view(), name='login'),
    path('logout', LogoutView.as_view(), name='logout'),
]
```

These views take care of logging a user in and out. However, the login view requires a template to render nicely. The `LoginView` expects it under the `registration/login.html` name.

We'll put our template in `django/user/templates/registration/login.html`:

```
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block title %} Login - {{ block.super }} {% endblock %}

{% block body %}
  <h1>Login</h1>
  <form method="post" class="col-sm-6">
    {% csrf_token %}
    {{ form|crispy }}
    <button type="submit" class="btn btn-primary">Login</button>
  </form>
{% endblock %}
```

The `LogoutView` doesn't require a template.

Now, we will need to inform our Django project's `settings.py` about the login view's location and the function it should perform when the user logs in and out. Let's add some settings to `django/config/settings.py`:

```
LOGIN_URL = 'user:login'
LOGIN_REDIRECT_URL = 'questions:index'
LOGOUT_REDIRECT_URL = 'questions:index'
```

This way, the `LoginRequiredMixin` can know the view to which we need to redirect unauthenticated users. We are also informing `LoginView` and `LogoutView` of `django.contrib.auth` where to redirect the user when they log in and log out, respectively.

Next, let's give users a way to register for our site.

Creating RegisterView

Django doesn't provide a user registration view, but it does offer a `UserCreationForm` if we're using `django.conrib.auth.models.User` as our user model. Since we are using `django.conrib.auth.models.User` we can use a simple `CreateView` for our registration view:

```
from django.contrib.auth.forms import UserCreationForm
from django.views.generic.edit import CreateView


class RegisterView(CreateView):
    template_name = 'user/register.html'
    form_class = UserCreationForm
```

Now, we just need to create a template at `django/user/templates/register.html`:

```
{% extends "base.html" %}
{% load crispy_forms_tags %}
{% block body %}
  <div class="col-sm-12">
    <h1 >Register for MyQA</h1 >
    <form method="post" >
      {% csrf_token %}
      {{ form | crispy }}
      <button type="submit" class="btn btn-primary" >
        Register
      </button >
    </form >
  </div >
{% endblock %}
```

Again, our template is following a familiar pattern similar what we've seen in past views. We use our base template, blocks, and Django Crispy Form to create our page quickly and simply.

Finally, we can add a `path` to the view in the `user` URLConf's `urlpatterns` list:

```
path('register', user.views.RegisterView.as_view(), name='register'),
```

Updating base.html navigation

Now that we have created all our views, we can update our base template's `<nav>` to list all our URLs:

```
{% load static %}
<!DOCTYPE html>
<html lang="en" >
<head >
{# skipping unchanged head contents #}
</head >
<body >
<nav class="navbar navbar-expand-lg  bg-light" >
  <div class="container" >
    <a class="navbar-brand" href="/" >Answerly</a >
    <ul class="navbar-nav" >
      <li class="nav-item" >
        <a class="nav-link" href="{% url "qanda:ask" %}" >Ask</a >
      </li >
      <li class="nav-item" >
        <a
            class="nav-link"
            href="{% url "qanda:index" %}" >
          Today's 🔥 Questions
        </a >
      </li >
      {% if user.is_authenticated %}
        <li class="nav-item" >
          <a class="nav-link" href="{% url "user:logout" %}" >Logout</a >
        </li >
      {% else %}
        <li class="nav-item" >
          <a class="nav-link" href="{% url "user:login" %}" >Login</a >
        </li >
        <li class="nav-item" >
          <a class="nav-link" href="{% url "user:register" %}" >Register</a
>
        </li >
      {% endif %}
    </ul >
  </div >
```

```
</nav >
<div class="container" >
  {% block body %}{% endblock %}
</div >
</body >
</html >
```

Great! Now our user can always reach the most important pages on our site.

Running the development server

Finally, we can access our development server using the following command:

```
$ cd django
$ python manage.py runserver
```

Now we can open the site in a browser at `http://localhost:8000/`.

Summary

In this chapter, we started our Answerly project. Answerly is composed of two apps (`user` and `qanda`), two third-party apps installed via PyPI (Markdownify and Django Crispy Forms), and a number of Django's built-in apps (`django.contrib.auth` being used most directly).

A logged-in user can now ask a question, answer questions, and accept answers. We can also see each day's highest-voted questions.

Next, we'll help users discover questions more easily by adding search functionality using ElasticSearch.

7
Searching for Questions with Elasticsearch

Now that users can ask and answer questions, we'll add a search functionality to Answerly to help users find questions. Our search will be powered by Elasticsearch. Elasticsearch is a popular open source search engine powered by Apache Lucene.

In the chapter, we will do the following things:

- Create an Elasticsearch service to abstract our code
- Bulk load existing `Question` model instances into Elasticsearch
- Build a search view powered by Elasticsearch
- Save new models into Elasticsearch automatically

Let's start by setting up our project to use Elasticsearch.

Starting with Elasticsearch

Elasticsearch is maintained by Elastic, though the server is open source. Elastic offers proprietary plugins to make running it in production easier. You can run Elasticsearch yourself or use a SaaS provider, such as Amazon, Google, or Elastic. In development, we'll run Elasticsearch using a Docker image provided by Elastic.

Elasticsearch is made up of zero or more indexes. Each index contains documents. Documents are the objects that one searches for. A document is made of up fields. Fields are indexed by Apache Lucene. Each index is also split up into one or more shards to make indexing and searching faster by distributing it across nodes in a cluster.

We can interact with Elasticsearch using its RESTful API. Most requests and responses are in JSON by default.

First, let's start by getting an Elasticsearch server running in Docker.

Starting an Elasticsearch server with docker

The simplest way to get an Elasticsearch server running is using the Docker image that Elastic provides.

To obtain and start the Elasticsearch docker image, run the following command:

```
$ docker run -d -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node"
docker.elastic.co/elasticsearch/elasticsearch:6.0.0
```

The following command does four things, as follows:

- It downloads the Elasticsearch 6.0 docker image from Elastic's servers
- It runs a container using the Elasticsearch 6.0 docker image as a single node cluster
- It detaches (`-d`) the docker command from the running container (so that we can run more commands in our shell)
- It opens ports (`-p`) `9200` and `9300` on the host computer and redirects them to the container

To confirm that our server is running, we can make the following request to the Elasticsearch server:

```
$ curl http://localhost:9200/?pretty
{
  "name" : "xgf60cc",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "HZAnjZefSjqDOxbMU99KOw",
  "version" : {
    "number" : "6.0.0",
    "build_hash" : "8f0685b",
    "build_date" : "2017-11-10T18:41:22.859Z",
    "build_snapshot" : false,
    "lucene_version" : "7.0.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

When interacting with Elasticsearch yourself, always add the `pretty` `GET` parameter to have Elasticsearch print the JSON. However, don't use this parameter in your code.

Now that we have our Elasticsearch server, let's configure Django to know about our server.

Configuring Answerly to use Elasticsearch

Next, we'll update our `settings.py` and `requirements.txt` files to work with Elasticsearch.

Let's update `django/config/settings.py`:

```
ES_INDEX = 'answerly'
ES_HOST = 'localhost'
ES_PORT = '9200'
```

These are custom settings that our app will use. Django has no built-in support for Elasticsearch. Instead, we'll reference these settings in our own code.

Let's add the Elasticsearch library to our `requirements.txt` file:

```
elasticsearch==6.0.0
```

This is the official Elasticsearch Python library published by Elastic. This library offers a low-level interface that looks much like the RESTful API we can use with cURL. This means that we can easily build a query on the command line with cURL and then convert the JSON to a Python `dict`.

Elastic also offers a higher-level, more Pythonic API called `elasticsearch-dsl`. It includes a pseudo-ORM to write a more Pythonic persistence layer. This may be a good option if your project includes a lot of Elasticsearch code. However, the low-level API closely mirrors the RESTful API, making it easier to reuse code and get assistance from the Elasticsearch community.

Next, let's create the Answerly index in our Elasticsearch server.

Creating the Answerly index

Let's create an index in Elasticsearch by sending a `PUT` request to our server:

```
$ curl -XPUT "localhost:9200/answerly?pretty"
```

Great! Now, we can load our existing `Question` model instances into our Elasticsearch index.

Loading existing Questions into Elasticsearch

Adding a search feature means that we will need to load our existing `Question` model instances into Elasticsearch. The simplest way to solve a problem like this is by adding a `manage.py` command. Custom `manage.py` commands combine the simplicity of writing a regular Python script with the power of the Django API.

Before we add our `manage.py` command, we will need to write our Elasticsearch-specific code. To separate the Elasticsearch code from our Django code, we'll add an `elasticsearch` service to the `qanda` app.

Creating the Elasticsearch service

Much of the code that we'll be writing in this chapter will be Elasticsearch specific. We don't want to put that code in our views (or `manage.py` commands) because that would introduce coupling between two unrelated components. Instead, we'll isolate the Elasticsearch code into its own module inside `qanda`, then have our views and `manage.py` command call our service module.

The first function we'll create will bulk load `Question` model instances into Elasticsearch.

Let's create a separate file for our Elastic Service code. We'll put our bulk insert code into `django/qanda/service/elasticsearch.py`:

```
import logging

from django.conf import settings
from elasticsearch import Elasticsearch, TransportError
from elasticsearch.helpers import streaming_bulk

FAILED_TO_LOAD_ERROR = 'Failed to load {}: {!r}'

logger = logging.getLogger(__name__)


def get_client():
    return Elasticsearch(hosts=[
        {'host': settings.ES_HOST, 'port': settings.ES_PORT,}
    ])


def bulk_load(questions):
    all_ok = True
    es_questions = (q.as_elasticsearch_dict() for q in questions)
    for ok, result in streaming_bulk(
            get_client(),
            es_questions,
            index=settings.ES_INDEX,
            raise_on_error=False,
    ):
        if not ok:
            all_ok = False
            action, result = result.popitem()
            logger.error(FAILED_TO_LOAD_ERROR.format(result['_id'],
result))
    return all_ok
```

We've created two functions in our new service, `get_client()` and `bulk_load()`.

The `get_client()` function will return an `Elasticcearch` client configured with values from `settings.py`.

The `bulk_load()` function takes an iterable collection of `Question` model instances and loads them into Elasticsearch using the `streaming_bulk()` helper. Since `bulk_load()` expects an iterable collection, this means that our `manage.py` command will be able to send a `QuerySet` object. Remember that even though we're using a generator expression (which is lazy), our `questions` parameter will execute the full query as soon as we try to iterate over it. It's only the execution of the `as_elasticsearch_dict()` method that will be lazy. We'll write and discuss the new `as_elasticsearch_dict()` method after we're finished looking at the `bulk_load()` function.

Next, the `bulk_load()` function uses the `streaming_bulk()` function. The `streaming_bulk()` function takes four arguments and returns an iterator for reporting the progress of the load. The four arguments are as follows:

- An `Elasticsearch` client
- Our `Question` generator (an iterator)
- The index name
- A flag telling the function not to raise an exception in case of an error (this will cause the `ok` variable to be `False` in case of errors)

The body of our `for` loop will log if there's an error when loading a question.

Next, let's give `Question` a method that can convert it into a `dict` that Elasticsearch can correctly process.

Let's update the `Question` model:

```
from django.db import models


class Question(models.Model):
    # fields and methods unchanged

    def as_elasticsearch_dict(self):
        return {
            '_id': self.id,
            '_type': 'doc',
            'text': '{}\n{}'.format(self.title, self.question),
            'question_body': self.question,
            'title': self.title,
            'id': self.id,
            'created': self.created,
        }
```

The `as_elasticsearch_dict()` method turns a `Question` model instance into a dict suitable for loading into Elasticsearch. The following are the three fields that we specially add to our Elasticsearch dict that aren't in our model:

- `_id`: This is the ID of the Elasticsearch document. This doesn't have to be the same as the model ID. However, if we want to be able to update the Elasticsearch document representing a `Question`, then we need to either store the document's `_id` or be able to calculate it. For simplicity's sake, we just use the same ID.
- `_type`: This is the document's mapping type. As of Elasticsearch 6, Elasticsearch indexes are only able to store one mapping type each. So, all documents in the index should have the same `_type` value. Mapping types are similar to database schema's, telling Elasticsearch how to index and track a document and its fields. One of the convenient features of Elasticsearch is that it doesn't require us to define our type ahead of time. Elasticsearch dynamically builds the document's type based on the data we load.
- `text`: This is a field we will create in the document. For search, it's convenient to have the title and body of the document together in an indexable field.

The rest of the fields in the dictionary are the same as the model's fields.

The presence of `as_elasticsearch_dict()` as a model method may seem problematic. Shouldn't the `elasticsearch` service know how to convert a `Question` in to an Elasticsearch dict? Like many design questions, the answer depends on a variety of factors. One factor that influenced me adding this method to the model is Django's *fat models* philosophy. Generally, Django encourages writing operations on the model as model methods. Also, the properties of this dict are coupled to the model's fields. Keeping both the lists of fields close together makes it easier for future developers to keep the two lists in sync. However, there may be projects and contexts in which the right thing is to put this kind of function in the service module. As Django developers, it's our job to evaluate the trade-offs and make the best decision for a particular project.

Now that our `elasticsearch` service knows how to bulk add `Questions`, let's expose that functionality with a `manage.py` command.

Creating a manage.py command

We've used `manage.py` commands to start projects and apps as well as create and run migrations. Now, we'll create a custom command to load all the questions in our project into an Elasticsearch server. This will be a simple introduction to Django management commands. We'll discuss the topic more in `Chapter 12`, *Building an API*.

A Django management command must be in an app's `manage/commands` subdirectory. An app may have multiple commands. Each command will be called the same as its filename. Inside the file should be a `Command` class that subclasses `django.core.management.BaseCommand`. The code that it should execute should be in the `handle()` method.

Let's create our command in `django/qanda/management/commands/load_questions_into_elastic_search.py`:

```
from django.core.management import BaseCommand

from qanda.service import elasticsearch
from qanda.models import Question


class Command(BaseCommand):
    help = 'Load all questions into Elasticsearch'

    def handle(self, *args, **options):
        queryset = Question.objects.all()
        all_loaded = elasticsearch.bulk_load(queryset)
        if all_loaded:
            self.stdout.write(self.style.SUCCESS(
                'Successfully loaded all questions into Elasticsearch.'))
        else:
            self.stdout.write(
                self.style.WARNING('Some questions not loaded '
                                   'successfully. See logged errors'))
```

When designing commands, we should think of them as views, that is, *Fat models, thin commands*. This may be a bit more complicated, as there isn't a separate template layer for command-line output, but our output shouldn't be very complex anyway.

In our case, the `handle()` method gets a `QuerySet` of all `Questions` then passes it to `elasticsearch.bulkload`. We then print out whether it was successful or not using helper methods of `Command`. These helper methods are preferred over using `print()` directly because they make writing tests easier. We'll cover this topic in greater detail in our next chapter, `Chapter 8`, *Testing Answerly*.

Let's run the following command:

```
$ cd django
$ python manage.py load_questions_into_elastic_search
Successfully loaded all questions into Elasticsearch.
```

With all the questions loaded, let's confirm that they're in our Elasticsearch server. We can access the Elasticsearch server using `curl` to confirm that our questions have been loaded:

```
$ curl http://localhost:9200/answerly/_search?pretty
```

Assuming your ElasticSearch server is running on localhost on port 9200, the preceding command will return all the data in the `answerly` index. We can review the results to confirm that our data has been successfully loaded.

Now that we have some questions in Elasticsearch, let's add a search view.

Creating a search view

In this section, we'll create a view that will let users search our `Question`s and will display the matching results. To achieve this result, we will do the following things:

- Add a `search_for_question()` function to our `elasticsearch` service
- Make a search view
- Make a template to display search results
- Update the base template to have search available everywhere

Let's start by adding search to our `elasticsearch` service.

Creating a search function

The responsibility for querying our Elasticsearch server for a list of questions matching the user's query lies with our `elasticsearch` service.

Let's add a function that will send a search query and parse the results to `django/qanda/service/elasticsearch.py`:

```
    def search_for_questions(query):
        client = get_client()
        result = client.search(index=settings.ES_INDEX, body={
          'query': {
              'match': {
                  'text': query,
              },
          },
        })
        return (h['_source'] for h in result['hits']['hits'])
```

After we connect with the client, we will send our query and parse the results.

Using the client's `search()` method, we send the query as a Python `dict` in the Elasticsearch Query DSL (domain-specific language). The Elasticsearch Query DSL provides a language for querying Elastic search using a series of nested objects. When sent by HTTP, the query becomes a series of nested JSON objects. In Python, we use `dict` s.

In our case, we're using a `match` query on the `text` field of the documents in the Answerly index. A `match` query is a fuzzy query that checks each document's `text` field to check whether it matches. The Query DSL also supports a number of configuration options to let you build more complex queries. In our case, we will accept the default fuzzy configuration.

Next, `search_for_questions` iterates over the results. Elasticsearch returns a lot of metadata describing the number of results, the quality to the match, and the resulting document. In our case, we will return an iterator of the matching documents (stored in`_source`).

Now that we can get our results from Elasticsearch, we can write our `SearchView`.

Creating the SearchView

Our `SearchView` will take a `GET` parameter `q` and perform a search using our service module's `search_for_questions()` function.

We'll build our `SearchView` using a `TemplateView`. `TemplateView` renders a template in response to `GET` requests. Let's add `SearchView` to `django/qanda/views.py`:

```
from django.views.generic import TemplateView

from qanda.service.elasticsearch import search_for_questions


class SearchView(TemplateView):
    template_name = 'qanda/search.html'

    def get_context_data(self, **kwargs):
        query = self.request.GET.get('q', None)
        ctx = super().get_context_data(query=query, **kwargs)
        if query:
            results = search_for_questions(query)
            ctx['hits'] = results
        return ctx
```

Next, we'll add a `path()` object routing to our `SearchView` to our URLConf in `django/qanda/urls.py`:

```
from django.urls.conf import path, include

from qanda import views

app_name = 'qanda'

urlpatterns = [
    # skipping previous code
    path('q/search', views.SearchView.as_view(),
         name='question_search'),
]
```

Now that we have our our view, let's build our `search.html` template.

Creating the search template

We'll put our search template in `django/qanda/templates/qanda/search.html`, as follows:

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

{% load markdownify %}
```

```
{% block body %}
  <h2 >Search</h2 >
  <form method="get" class="form-inline" >
    <input class="form-control mr-2"
           placeholder="Search"
           type="search"
           name="q" value="{{ query }}" >
    <button type="submit" class="btn btn-primary" >Search</button >
  </form >
  {% if query %}
    <h3>Results from search query '{{ query }}'</h3 >
    <ul class="list-unstyled search-results" >
      {% for hit in hits %}
        <li >
          <a href="{% url "qanda:question_detail" pk=hit.id %}" >
            {{ hit.title }}
          </a >
          <div >
            {{ hit.question_body|markdownify|truncatewords_html:20 }}
          </div >
        </li >
      {% empty %}
        <li >No results.</li >
      {% endfor %}
    </ul >
  {% endif %}
{% endblock %}
```

In the body of the template, we have a search form that displays the query. If there was a `query`, then we will also show its results (if any).

We have seen many of the tags we are using here before (for example, `for`, `if`, `url`, and `markdownify`). A new filter that we will add is `truncate_words_html`, which receives text via the pipe and a number as an argument. It will truncate the text to the provided number of words (not counting HTML tags) and close any open HTML tags in the resulting fragment.

The result of this template is a list of hits that match our query with a preview of the text of each question. Since we stored the body, title, and ID of the question in Elasticsearch, we are able to show the results without querying our normal database.

Next, let's update our base template to let users search from every page.

Updating the base template

Let's update the base template to let users search from anywhere. To do that, we'll need to edit `django/templates/base.html`:

```
{% load static %}
<!DOCTYPE html>
<html lang="en" >
<head >{# head unchanged #}</head >
<body >
<nav class="navbar navbar-expand-lg  bg-light" >
  <div class="container" >
    <a class="navbar-brand" href="/" >Answerly</a >
    <ul class="navbar-nav" >
      {# previous nav unchanged #}
      <li class="nav-item" >
        <form class="form-inline"
              action="{% url "qanda:question_search" %}"
              method="get">
          <input class="form-control mr-sm-2" type="search"
                 name="q"
                 placeholder="Search">
          <button class="btn btn-outline-primary my-2 my-sm-0"
                 type="submit" >
            Search
          </button >
        </form >
      </li >
    </ul >
  </div >
</nav >
{# rest of body unchanged #}
</body >
</html >
```

Now, we've got the search form in our header on every page.

With our search complete, let's make sure that every new question is automatically added to Elasticsearch.

Adding Questions into Elasticsearch on save()

The best way to perform an operation that is each time a model is saved to override the `save()` method that the model inherits from `Model`. We will provide a custom `Question.save()` method to make sure that `Question`s are added and updated in ElasticSearch as soon as they're saved by the Django ORM.

You can still perform an operation when a Django model is saved even if you don't control the source code of that model. Django offers a signals dispatcher (`https://docs.djangoproject.com/en/2.0/topics/signals/`) that lets you listen for events on models you don't own. However, signals introduce a lot of complexity into your code. It's *discouraged* to use signals unless there is no other option.

Let's update our `Queston` model in `django/qanda/models.py`:

```
from django.db import models
from qanda.service import elasticsearch
class Question(models.Model):
    # other fields and methods unchanged.
    def save(self, force_insert=False, force_update=False, using=None,
             update_fields=None):
        super().save(force_insert=force_insert,
                     force_update=force_update,
                     using=using,
                     update_fields=update_fields)
        elasticsearch.upsert(self)
```

The `save()` method is called by `CreateView`, `UpdateView`, `QuerySet.create()`, `Manager.create()`, and most third-party code to persist a model. We make sure to call our `upsert()` method after the original `save()` method has returned because we want our model to have an `id` attribute.

Now, let's create our Elasticsearch service's `upsert` method.

Upserting into Elasticsearch

An upsert operation will update an object if it exists and insert it doesn't. Upsert is a portmanteau of *update* and *insert*. Elasticsearch supports upsert operations out of the box, which can make our code much simpler.

Let's add our `upsert()` method to `django/qanda/service/elastic_search.py`:

```
def upsert(question_model):
    client = get_client()
    question_dict = question_model.as_elasticsearch_dict()
    doc_type = question_dict['_type']
    del question_dict['_id']
    del question_dict['_type']
    response = client.update(
        settings.ES_INDEX,
        doc_type,
        id=question_model.id,
        body={
            'doc': question_dict,
            'doc_as_upsert': True,
        }
    )
    return response
```

We've defined our `get_client()` function in the preceding code block.

To perform an upsert, we use the `update()` method of Elasticsearch `client`. We provide the model as a document `dict` under the `doc` key. To force Elasticsearch to perform an upsert, we will include the `doc_as_upsert` key with a `True` value. One difference between the `update()` method and the bulk insert function we used earlier is that `update()` will not accept an implicit ID (`_id`) in the document. However, we provide the ID of the document to upsert as the `id` argument in our `update()` call. We also remove the `_type` key and value from the `dict` returned by the `question_model.as_elasticsearch_dict()` method and pass value (stored in the `doc_type` variable) as an argument to the `client.update()` method.

We return the response, though our view won't use it.

Finally, we can test our view by running our development server:

```
$ cd django
$ python manage.py runserver
```

Once our development server has started, we can ask a new question at `http://localhost:8000/ask` and then search for it at `http://localhost:8000/q/search`.

Now, we're done adding search functionalities to Answerly!

Summary

In this chapter, we've added search so that users can search for questions. We set up an Elasticsearch server for development using Docker. We created a `manage.py` command to load all our `Question`s into Elasticsearch. We added a search view where users could see the results of their question. Finally, we updated `Question.save` to keep Elasticsearch and the Django DB in sync.

Next, we'll take an in-depth look at testing a Django app so that we can have confidence as we make future changes.

8
Testing Answerly

In the preceding chapter, we added search to Answerly, our question and answer site. However, as our site's functionality grows, we need to avoid breaking the existing functionality. To make sure that our code keeps working, we will take a closer look at testing our Django project.

In this chapter, we will do the following things:

- Install Coverage.py to measure code coverage
- Measure code coverage for our Django project
- Write a unit test for our model
- Write a unit test for a view
- Write a Django integration tests for a view
- Write a Selenium integration test for a view

Let's start by installing Coverage.py.

Installing Coverage.py

Coverage.py is the most popular Python code coverage tool at the time of writing. It's very easy to install as it's available from PyPI. Let's add it to our `requirements.txt` file:

```
$ echo "coverage==4.4.2" >> requirements.txt
```

Then we can install Coverage.py using pip:

```
$ pip install -r requirements.txt
```

Now that we have Coverage.py installed, we can start measuring our code coverage.

Measuring code coverage

Code coverage measures which lines of code have been executed during a test. Ideally, by tracking code coverage, we can ensure which code is tested and which code is not. Since Django projects are mainly Python, we can use Coverage.py to measure our code coverage. The following are the two caveats for Django projects:

- Coverage.py won't be able to measure the coverage of our templates (they're not Python)
- Untested class-based views seem more covered than they are

Finding the coverage of a Django app is a two-step process:

1. Running our tests with the `coverage` command
2. Generating a coverage report using `coverage report` or `coverage html`

Let's run Django's unit `test` command with `coverage` to take a look at the baseline for an untested project:

```
$ coverage run --branch --source=qanda,user manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
Destroying test database for alias 'default'...
```

The preceding command tells `coverage` to run a command (in our case, `manage.py test`) to record test coverage. We will use this command with the following two options:

- `--branch`: To track whether both parts of branching statements were covered (for example, when an `if` statement evaluated to `True` and `False`)
- `--source=qanda,user`: To record coverage only for the `qanda` and `user` modules (the code we wrote)

Now that we've recorded the coverage, let's take a look at the coverage of an app without any tests:

```
$ coverage report
Name                                    Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------------
qanda/__init__.py                           0      0      0      0   100%
qanda/admin.py                              1      0      0      0   100%
qanda/apps.py                               3      3      0      0     0%
qanda/forms.py                             19      0      0      0   100%
qanda/management/__init__.py                0      0      0      0   100%
qanda/migrations/0001_initial.py            7      0      0      0   100%
qanda/migrations/__init__.py                0      0      0      0   100%
qanda/models.py                            28      6      0      0    79%
qanda/search_indexes.py                     0      0      0      0   100%
qanda/service/__init__.py                   0      0      0      0   100%
qanda/service/elasticsearch.py             47     32     14      0    25%
qanda/tests.py                              1      0      0      0   100%
qanda/urls.py                               4      0      0      0   100%
qanda/views.py                             76     35     12      0    47%
user/__init__.py                            0      0      0      0   100%
user/admin.py                               4      0      0      0   100%
user/apps.py                                3      3      0      0     0%
user/migrations/__init__.py                 0      0      0      0   100%
user/models.py                              1      0      0      0   100%
user/tests.py                               1      0      0      0   100%
user/urls.py                                5      0      0      0   100%
user/views.py                               5      0      0      0   100%
-------------------------------------------------------------------------
TOTAL                                     205     79     26      0    55%
```

To understand how an untested project is 55% covered, let's look at the coverage of `django/qanda/views.py`. Let's generate an HTML report of the cover using the following command:

```
$ cd django
$ coverage html
```

The preceding command will create a `django/htmlcov` directory and HTML files that show the coverage report and a visual display of the code coverage. Let's open `django/htmlcov/qanda_views_py.html` and scroll down to around line 72:

```
72  class DailyQuestionList(DayArchiveView):
73      queryset = Question.objects.all()
74      date_field = 'created'
75      month_format = '%m'
76      allow_empty = True
77
78
79  class QuestionDetailView(DetailView):
80      model = Question
81
82      ACCEPT_FORM = AnswerAcceptanceForm(initial={'accepted': True})
83      REJECT_FORM = AnswerAcceptanceForm(initial={'accepted': False})
84
85      def get_context_data(self, **kwargs):
86          ctx = super().get_context_data(**kwargs)
87          ctx.update({
88              'answer_form': AnswerForm(initial={
89                  'user': self.request.user.id,
90                  'question': self.object.id,
91              })
92          })
93          if self.object.can_accept_answers(self.request.user):
94              ctx.update({
95                  'accept_form': self.ACCEPT_FORM,
96                  'reject_form': self.REJECT_FORM,
97              })
98          return ctx
99
100
```

The preceding screenshot shows that `DailyQuestionList` is completely covered but `QuestionDetailView.get_context_data()` is not. In the absence of any tests, the difference seems counterintuitive.

Let's remind ourselves how code coverage works. Code coverage tools check whether a particular line of code was *executed* during a test. In the preceding screenshot, the `DailyQuestionList` class and its members *were* executed. When the test runner starts, Django will build up the root URLConf much like when it starts for development or production. When the root URLConf is created, it imports the other referenced URLConfs (for example, `qanda.urls`). Those URLConfs, in turn, import their views. Views import forms, models, and other modules.

This import chain means that anything at the top level of a module will appear covered, regardless of whether it is tested. The class definition of `DailyQuestionList` was executed. However, the class itself was not instantiated, nor any of its methods executed. This also explains why the body of `QuestionDetailView.get_context_data()` is not covered. The body of `QuestionDetailView.get_context_data()` was never executed. This is a limitation of code coverage tools when working with declarative code such as `DailyQuestionList`.

Now that we understand some of the limitations of code coverage, let's write a unit test for `qanda.models.Question.save()`.

Creating a unit test for Question.save()

Django helps you write unit tests to test individual units of code. If our code relies on an external service, then we can use the standard `unittest.mock` library to mock that API, preventing requests to outside systems.

Let's write a test for the `Question.save()` method to verify that when we save a `Question` it will be upserted into Elasticsearch. We'll write the test in `django/qanda/tests.py`:

```
from unittest.mock import patch

from django.conf import settings
from django.contrib.auth import get_user_model
from django.test import TestCase
from elasticsearch import Elasticsearch

from qanda.models import Question


class QuestionSaveTestCase(TestCase):
    """
    Tests Question.save()
    """

    @patch('qanda.service.elasticsearch.Elasticsearch')
    def test_elasticsearch_upsert_on_save(self, ElasticsearchMock):
        user = get_user_model().objects.create_user(
            username='unittest',
            password='unittest',
        )
        question_title = 'Unit test'
```

```
question_body = 'some long text'
q = Question(
    title=question_title,
    question=question_body,
    user=user,
)
q.save()

self.assertIsNotNone(q.id)
self.assertTrue(ElasticsearchMock.called)
mock_client = ElasticsearchMock.return_value
mock_client.update.assert_called_once_with(
    settings.ES_INDEX,
    id=q.id,
    body={
        'doc': {
            '_type': 'doc',
            'text': '{}\n{}'.format(question_title, question_body),
            'question_body': question_body,
            'title': question_title,
            'id': q.id,
            'created': q.created,
        },
        'doc_as_upsert': True,
    }
)
```

In the preceding code sample, we created a `TestCase` with a single test method. The method creates a user, saves a new `Question`, and then asserts that the mock has behaved correctly.

Like most `TestCase` s, `QuestionSaveTestCase` uses both Django's testing API and code from Python's `unittest` library (for example, `unittest.mock.patch()`). Let's look more closely at how Django's testing API makes testing easier.

`QuestionSaveTestCase` extends `django.test.TestCase` instead of `unittest.TestCase` because Django's `TestCase` offers lots of useful features, as follows:

- The entire test case and each test are atomic database operations
- Django takes care of clearing the database before and after each test
- `TestCase` offers convenient `assert*()` methods such as `self.assertInHTML()` (discussed more in the *Creating a unit test for a view* section)
- A fake HTTP client to create integration tests (discussed more in the *Creating an integration test for a view* section)

Since Django's `TestCase` extends `unittest.TestCase`, it still understands and performs correctly when it hits a regular `AssertionError`. So, if `mock_client.update.assert_called_once_with()` raises an `AssertionError` exception, Django's test runner knows how to handle it.

Let's run our tests with `manage.py`:

```
$ cd django
$ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.094s

OK
Destroying test database for alias 'default'...
```

Now that we know how to test a model, we can move on to testing views. As we're testing our views, though, we will need to create model instances. Using model's default managers to create model instances will become too verbose. Next, let's make it easier to create the models necessary for testing with Factory Boy.

Creating models for tests with Factory Boy

In our preceding test, we made a `User` model using `User.models.create_user`. However, that required us to provide a username and password, neither of which we really cared about. We just need a user, not a particular user. For many of our tests, the same principle will hold true for `Question` s and `Answer` s. The Factory Boy library will help us concisely create models in tests.

Factory Boy is particularly useful for Django developers because it knows how to create models based from Django `Model` classes.

Let's install Factory Boy:

```
$ pip install factory-boy==2.9.2
```

In this section, we'll use Factory Boy to create a `UserFactory` class and a `QuestionFactory` class. Since a `Question` model must have a user in its `user` field, the `QuestionFactory` will show us how `Factory` classes can reference each other.

Let's start with the `UserFactory`.

Creating a UserFactory

Both `Question` s and `Answer` s are related to users. This means that we'll need to create users in almost all our tests. Generating all the related models for each test using model managers is very verbose and distracting from point of our tests. Django offers an out-of-the-box support for fixtures of our tests. However, Django's fixtures are separate JSON/YAML files that need to be manually maintained or they will grow out of sync and cause problems. Factory Boy will help us by letting use code, a `UserFactory` that can concisely create users model instances at runtime based on the state of the current User model.

Our `UserFactory` will be derived from Factory Boy's `DjangoModelFactory` class, which knows how to deal with Django models. We'll use an inner `Meta` class to tell `UserFactory` which model it's creating (note how this is similar to the `Form` API). We'll also add class attributes to tell Factory Boy how to set values of the model's fields. Finally, we'll override the `_create` method to make `UserFactory` use the manager's `create_user()` method instead of the default `create()` method.

Let's create our `UserFactory` in `django/users/factories.py`:

```
from django.conf import settings

import factory


class UserFactory(factory.DjangoModelFactory):
    username = factory.Sequence(lambda n: 'user %d' % n)
    password = 'unittest'

    class Meta:
        model = settings.AUTH_USER_MODEL

    @classmethod
    def _create(cls, model_class, *args, **kwargs):
        manager = cls._get_manager(model_class)
        return manager.create_user(*args, **kwargs)
```

The `UserFactory` subclasses the `DjangoModelFactory`. The `DjangoModelFactory` will look at our class's `Meta` inner class (which follows the same pattern as `Form` classes).

Let's take a closer look at attributes of `UserFactory`:

- `password = 'unittest'`: This sets the password for each user to be of the same value.
- `username = factory.Sequence(lambda n: 'user %d' % n):Sequence` sets a different value for a field each time the factory creates a model. `Sequence()` takes callable, passes it however many times the factory has been used, and use the callable's return value as the new instance's field value. In our case, our users will have usernames such as `user 0` and `user 1`.

Finally, we overrode the `_create()` method because the `django.contrib.auth.models.User` model has an unusual manager. The default `_create` method of `DjangoModelFactory` will use the model's manager's `create()` method. This is fine for most models, but won't work well for the `User` model. To create a user, we should really use the `create_user` method so that we can pass a password in plain text and have it hashed for storage. This will let us authenticate as that `User`.

Let's try out our factory using the Django shell:

```
$ cd django
$ python manage.py shell
Python 3.6.3 (default, Oct 31 2017, 11:15:24)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
In [1]: from user.factories import UserFactory
In [2]:  user = UserFactory()
In [3]: user.username
Out[3]: 'user 0'
In [4]:  user2 = UserFactory()
In [5]:  assert user.username != user2.username
In [6]: user3 = UserFactory(username='custom')
In [7]: user3.username
Out[7]: 'custom'
```

In this Django shell session, we will note how we can use `UserFactory`:

- We can create new models using a single no-argument call, `UserFactory()`
- Each call leads to a unique username, `assert user.username != user2.username`
- We can change values the factory used by providing them as arguments, `UserFactory(username='custom')`

Next, let's create a `QuestionFactory`.

Creating the QuestionFactory

Lots of our tests will require multiple `Question` instances. However, each `Question` must have a user. This can lead to lots of brittle and verbose code. Creating a `QuestionFactory` will solve this problem.

In the preceding example, we saw how we can use `factory.Sequence` to give each new model's attribute a distinct value. Factory Boy also offers `factory.SubFactory`, in which we can indicate that a field's value is the result of another factory.

Let's add `QuestionFactory` to `django/qanda/factories.py`:

```
from unittest.mock import patch

import factory

from qanda.models import Question
from user.factories import UserFactory

class QuestionFactory(factory.DjangoModelFactory):
    title = factory.Sequence(lambda n: 'Question #%d' % n)
    question = 'what is a question?'
    user = factory.SubFactory(UserFactory)

    class Meta:
        model = Question

    @classmethod
    def _create(cls, model_class, *args, **kwargs):
        with patch('qanda.service.elasticsearch.Elasticsearch'):
            return super()._create(model_class, *args, **kwargs)
```

Our `QuestionFactory` is very similar to our `UserFactory`. They have the following things in common:

- Derived from the `factory.DjangoModelFactory`
- Have a `Meta` class
- Use `factory.Sequence` to give a field a custom value
- Have a hardcoded value

There are two important differences:

- The `user` field of `QuestionFactory` uses `SubFactory` to give each `Question` a new user created with the `UserFactory`.
- The `_create` method of `QuestionFactory` mocks the Elasticsearch service so that when the model is created, it doesn't try to connect to that service. Otherwise, it calls the default `_create()` method.

To see our `QuestionFactory` in practice, let's write a unit test for our `DailyQuestionList` view.

Creating a unit test for a view

In this section, we'll write a view unit test for our `DailyQuestionList` view.

Unit testing a view means directly passing the view a request and asserting that the response matches our expectations. Since we're passing the request directly to the view, we also need to directly pass any arguments the view would ordinarily receive parsed out of the request's URL. Parsing values out of URL paths is the responsibility of the request routing, which we don't use in a view unit test.

Let's take a look at our `DailyQuestionListTestCase` class in `django/qanda/tests.py`:

```
from datetime import date

from django.test import TestCase, RequestFactory

from qanda.factories import QuestionFactory
from qanda.views import DailyQuestionList

QUESTION_CREATED_STRFTIME = '%Y-%m-%d %H:%M'


class DailyQuestionListTestCase(TestCase):
"""
Tests the DailyQuestionList view
"""
QUESTION_LIST_NEEDLE_TEMPLATE = '''
<li >
    <a href="/q/{id}" >{title}</a >
    by {username} on {date}
</li >
'''
```

```
    REQUEST = RequestFactory().get(path='/q/2030-12-31')
    TODAY = date.today()

    def test_GET_on_day_with_many_questions(self):
        todays_questions = [QuestionFactory() for _ in range(10)]

        response = DailyQuestionList.as_view()(
            self.REQUEST,
            year=self.TODAY.year,
            month=self.TODAY.month,
            day=self.TODAY.day
        )

        self.assertEqual(200, response.status_code)
        self.assertEqual(10, response.context_data['object_list'].count())
        rendered_content = response.rendered_content
        for question in todays_questions:
            needle = self.QUESTION_LIST_NEEDLE_TEMPLATE.format(
                id=question.id,
                title=question.title,
                username=question.user.username,
                date=question.created.strftime(QUESTION_CREATED_STRFTIME)
            )
            self.assertInHTML(needle, rendered_content)
```

Let's take a closer look at the new APIs we've seen:

- `RequestFactory().get(path=...)`: `RequestFactory` is a utility for creating HTTP requests for testing views. Note that our request's `path` is arbitrary here, as it won't be used for routing.
- `DailyQuestionList.as_view()(...)`: We've discussed that each class-based view has an `as_view()` method that returns a callable, but we haven't used it before. Here, we pass in the request, year, month, and day to execute the view.
- `response.context_data['object_list'].count()`: The response returned by our view still has its context. We can use this context to assert whether the view worked correctly more easily than if we had to evaluate the HTML.
- `response.rendered_content`: The `rendered_content` property lets us access the rendered template of the response.

- `self.assertInHTML(needle, rendered_content)`: `TestCase.assertInHTML()` lets us assert whether one HTML fragment is inside another. `assertInHTML()` knows how to parse HTML and doesn't care about attribute order or whitespace. In testing views, we frequently have to check whether a particular bit of HTML is present in a response.

Now that we've created a unit test for a view, let's look at creating an integration test for a view by creating an integration test for `QuestionDetailView`.

Creating a view integration test

View integration tests use the same `django.test.TestCase` class that a unit test does. An integration test will tell us if our project can route the request to the view and return the correct response. An integration test request will have to go through all the middleware and URL routing that a project is configured with. To help us write integration tests, Django provides `TestCase.client`.

`TestCase.client` is a utility offered by `TestCase` to let us send HTTP requests to our project (it can't send external HTTP requests). Django processes these requests normally. `client` also offers us convenience methods such as `client.login()`, a way of starting an authenticated session. A `TestCase` class also resets its `client` between each test.

Let's write an integration test for `QuestionDetailView` in `django/qanda/tests.py`:

```
from django.test import TestCase

from qanda.factories import QuestionFactory
from user.factories import UserFactory

QUESTION_CREATED_STRFTIME = '%Y-%m-%d %H:%M'


class QuestionDetailViewTestCase(TestCase):
    QUESTION_DISPLAY_SNIPPET = '''
    <div class="question" >
      <div class="meta col-sm-12" >
        <h1 >{title}</h1 >
        Asked by {user} on {date}
      </div >
      <div class="body col-sm-12" >
        {body}
      </div >
```

```
</div >'''
LOGIN_TO_POST_ANSWERS = 'Login to post answers.'


def test_logged_in_user_can_post_answers(self):
    question = QuestionFactory()

    self.assertTrue(self.client.login(
        username=question.user.username,
        password=UserFactory.password)
    )
    response = self.client.get('/q/{}'.format(question.id))
    rendered_content = response.rendered_content

    self.assertEqual(200, response.status_code)

     self.assertInHTML(self.NO_ANSWERS_SNIPPET, rendered_content)

    template_names = [t.name for t in response.templates]
    self.assertIn('qanda/common/post_answer.html', template_names)

    question_needle = self.QUESTION_DISPLAY_SNIPPET.format(
        title=question.title,
        user=question.user.username,
        date=question.created.strftime(QUESTION_CREATED_STRFTIME),
        body=QuestionFactory.question,
    )
    self.assertInHTML(question_needle, rendered_content)
```

In this sample, we log in and then request a detail view of `Question`. We make multiple assertions about the result to confirm that it is correct (including checking the name of the templates used).

Let's examine some of this code in greater detail:

- `self.client.login(...)`: This begins an authenticated session. All future requests will be authenticated as that user until we call `client.logout()`.
- `self.client.get('/q/{}'.format(question.id))`: This makes an HTTP `GET` request using our client. Unlike when we used `RequestFactory`, the path we provide is to route our request to a view (note that we never reference the view directly in the test). This returns the response created by our view.

- `[t.name for t in response.templates]`: When one of the client's responses renders, the client updates the response with a list of templates used. In the case of the detail view, we used multiple templates. In order to check whether we're showing the UI for posting an answer, we will check whether the `qanda/common/post_answer.html` file is one of the templates used.

With this kind of test, we can gain a lot of confidence that our view works when a user makes a request. However, it does couple the test to the project's configuration. Integration tests make sense even for views coming from third-party apps to confirm that they're being used correctly. If you're making an app that is a library, you may find it better to use a unit test.

Next, let's look at testing that our Django and frontend code are both working correctly by testing and creating a live server test case using Selenium.

Creating a live server integration test

The final type of test we'll write is a live server integration test. In this test, we'll start up a test Django server and make requests to it using Google Chrome controlled by Selenium.

Selenium is a tool with bindings for many languages (including Python) that lets you control a web browser. This lets you test exactly how a real browser behaves when it's using your project, because you are testing your project with a real browser.

There are some limitations imposed by this kind of test:

- Live tests often have to run in sequence
- It's easy to leak state across tests
- Using a browser is much slower than `TestCase.client()` (the browser makes real HTTP requests)

Despite all these downsides, live server tests can be an invaluable tool at a time when the client side of a web app is so powerful.

Let's start by setting up Selenium.

Setting up Selenium

Let's add Selenium to our project by installing with `pip`:

```
$pip install selenium==3.8.0
```

Next, we will need the particular webdriver that tells Selenium how to talk to Chrome. Google provides a **chromedriver** at `https://sites.google.com/a/chromium.org/chromedriver/`. In our case, let's save it at the root of our project directory. Then, let's add the path to that driver in `django/conf/settings.py`:

```
CHROMEDRIVER = os.path.join(BASE_DIR, '../chromedriver')
```

Finally, make sure that you have Google Chrome installed on your computer. If not, you can download it at `https://www.google.com/chrome/index.html`.

All major browsers claim to have some level of support for Selenium. If you don't like Google Chrome, you can try one of the others. Refer to Selenium's docs (`http://www.seleniumhq.org/about/platforms.jsp`) for details.

Testing with a live Django server and Selenium

Now that we have Selenium set up, we can create our live server test. A live server test is particularly useful when our project has a lot of JavaScript. Answerly, though, doesn't have any JavaScript. However, Django's forms do take advantage of HTML5 form attributes that most browsers (including Google Chrome) support. We can still test whether that functionality is being correctly used by our code.

In this test, we will check whether a user can submit an empty question. The `title` and `question` fields should each be marked `required` so that a browser won't submit the form if those fields are empty.

Let's add a new test to `django/qanda/tests.py`:

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase

from selenium.webdriver.chrome.webdriver import WebDriver

from user.factories import UserFactory


class AskQuestionTestCase(StaticLiveServerTestCase):
```

```
    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls.selenium = WebDriver(executable_path=settings.CHROMEDRIVER)
        cls.selenium.implicitly_wait(10)

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super().tearDownClass()

    def setUp(self):
        self.user = UserFactory()

    def test_cant_ask_blank_question(self):
        initial_question_count = Question.objects.count()

        self.selenium.get('%s%s' % (self.live_server_url, '/user/login'))

        username_input = self.selenium.find_element_by_name("username")
        username_input.send_keys(self.user.username)
        password_input = self.selenium.find_element_by_name("password")
        password_input.send_keys(UserFactory.password)
        self.selenium.find_element_by_id('log_in').click()

        self.selenium.find_element_by_link_text("Ask").click()
        ask_question_url = self.selenium.current_url
        submit_btn = self.selenium.find_element_by_id('ask')
        submit_btn.click()
        after_empty_submit_click = self.selenium.current_url

        self.assertEqual(ask_question_url, after_empty_submit_click)
        self.assertEqual(initial_question_count, Question.objects.count())
```

Let's take a look at some of the new Django features introduced in this test. Then, we'll review our Selenium code:

- `class AskQuestionTestCase(StaticLiveServerTestCase):StaticLiveServerTestCase` starts a Django server and also ensures that static files are served correctly. You don't have to run `python manage.py collectstatic`. The files will be routed correctly just like if you're running `python manage.py runserver`.

- `def setUpClass(cls)`: All the Django test cases support the `setUpClass()`, `setup()`, `teardown()`, and `teardownClass()` methods as usual. `setUpClass` and `tearDownClass()` are run only once per `TestCase` (before and after, respectively). This makes them ideal for expensive operations, such as connecting to Google Chrome with Selenium.
- `self.live_server_url`: This is the URL to the live server.

Selenium lets us interact with a browser using an API. This book is not focused on Selenium, but let's cover some key methods of the `WebDriver` class:

- `cls.selenium = WebDriver(executable_path=settings.CHROMEDRIVER)`: This instantiates a WebDriver instance with the path to the `ChromeDriver` executable (that we downloaded in the preceding *Setting Up Selenium* section). We stored the path to the `ChromeDriver` executable in our settings to let us easily reference it here.
- `selenium.find_element_by_name(...)`: This returns an HTML element whose `name` attribute matches the provided argument. `name`s attributes are used by all `<input>` elements whose value is processed by a form, so this is particularly useful for data entry.
- `self.selenium.find_element_by_id(...)`: This is like the preceding step, except find the matching element by its `id` attribute.
- `self.selenium.current_url`: This is the browser's current URL. This is useful for confirming that we're on the page we expect.
- `username_input.send_keys(...)`: The `send_keys()` method lets us type the passed string into the an HTML element. This is particularly useful for `<input type='text'>`and `<input type='password'>` elements.
- `submit_btn.click()`: This triggers a click on the element.

This test logs in as a user, tries to submit a form, and asserts that it is still on the same page. Unfortunately, while a form with an empty required `input` elements won't submit itself, there is no API to confirm that directly. Instead, we confirm that we haven't submitted because the browser is still at the same URL (according to `self.selenium.current_url`) as before we hit submit.

Summary

In this chapter, we learned how to measure code coverage in Django projects and how to write four different types of tests—unit tests for testing any function or class, including models and forms; and view unit tests for testing views using `RequestFactory`. We covered how to view integration tests for testing that request route to a view and return correct responses and Live server integration tests for testing that your client and server-side code work together correctly.

Now that we have some tests, let's deploy Answerly into a production environment.

9
Deploying Answerly

In the preceding chapter, we learned about Django's testing API and wrote some tests for Answerly. As the final step, let's deploy Answerly on an Ubuntu 18.04 (Bionic Beaver) server using the Apache web server and mod_wsgi.

This chapter assumes that you have the code on your server under `/answerly` and are able to push updates to that code. You will make some changes to your code in this chapter. Despite making changes, you will need to avoid developing the habit of making direct changes in production. For example, you might be using a version control system (such as git) to track changes in your code. Then, you can make changes on your local workstation, push them to a remote repository (for example, hosted on GitHub or GitLab), and pull them on your server. This code is available in version control on GitHub (`https://github.com/tomarayn/Answerly`).

In this chapter, we will do the following things:

- Organize our configuration code to separate production and development settings
- Prepare our Ubuntu Linux server
- Deploy our project using Apache and mod_wsgi
- Take a look at how Django lets us deploy our projects as twelve-factor apps

Let's start by organizing our configuration to separate development and production settings.

Organizing configuration for production and development

Up until now, we've kept a single `requirements` file and a single `settings.py`. This has made development convenient. However, we can't use our development settings in production.

The current best practice is to have a separate file for each environment. Each environment's file then imports a common file with shared values. We'll use this pattern for our requirements and settings files.

Let's start by splitting up our requirements file.

Splitting our requirements file

First, let's create `requirements.common.txt` at the root of our project:

```
django<2.1
psycopg2==2.7.3.2
django-markdownify==0.2.2
django-crispy-forms==1.7.0
elasticsearch==6.0.0
```

Regardless of our environment, these are our common requirements, which we'll need to run Answerly. However, this `requirements` file is never used directly. Our development and production requirements files will reference.

Next, let's list our development requirements in `requirements.development.txt`:

```
-r requirements.common.txt
ipython==6.2.1
coverage==4.4.2
factory-boy==2.9.2
selenium==3.8.0
```

The preceding file will install everything from `requirements.common.txt` (thanks to `-r`) as well as our testing packages (`coverage`, `factory-boy`, and `selenium`). We're putting these files in our development file because we don't expect to run these tests in our production environment. If we were running tests in production, then we'd probably move them to `requirements.common.txt`.

For production, our `requirements.production.txt` file is very simple:

```
-r requirements.common.txt
```

Answerly doesn't need any special packages. However, we will still create one for clarity.

To install packages in production, we now execute the following command:

```
$ pip install -r requirements.production.txt
```

Next, let's split up the settings file along similar lines.

Splitting our settings file

Again, we will follow the current Django best practice of splitting our settings file into three files: `common_settings.py`, `production_settings.py`, and `dev_settings.py`.

Creating common_settings.py

We'll create `common_settings.py` by renaming our current `settings.py` file and then making some changes.

Let's change `DEBUG = False` so that no new settings file can *accidentally* be in debug mode. Then, let's change the secret key to be obtained from an environment variable by updating `SECRET_KEY = os.getenv('DJANGO_SECRET_KEY')`.

Let's also add a new setting, `STATIC_ROOT`. `STATIC_ROOT` is the directory where Django will collect all the static files from across our installed apps to make it easier to serve them:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static_root')
```

In the database config, we can remove all the credentials and keep the `ENGINE` value (to make it clear that we intend to use Postgres everywhere):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
    }
}
```

Next, let's create a development settings file.

Creating dev_settings.py

Our development settings will be in `django/config/dev_settings.py`. Let's build it incrementally.

First, we will import everything from `common_settings`:

```
from config.common_settings import *
```

Then, we'll override some settings:

```
DEBUG = True
SECRET_KEY = 'some secret'
```

In development, we always want to run in debug mode. Also, we can feel safe hardcoding a secret key, as we know it won't be used in production:

```
DATABASES['default'].update({
    'NAME': 'mymdb',
    'USER': 'mymdb',
    'PASSWORD': 'development',
    'HOST': 'localhost',
    'PORT': '5432',
})
```

Since our development database is local, we can hardcode the values in our settings to make the settings simpler. If your database is not local, avoid checking passwords into version control and use `os.getenv()` like in production.

We can also add more settings that our development-only apps may require. For example, in `Chapter 5`, *Deploying with Docker*, we had settings for caches and the Django Debug Toolbar app. Answerly doesn't use those right now, so we won't include those settings.

Next, let's add production settings.

Creating production_settings.py

Let's create our production settings in `django/config/production_settings.py`.

`production_settings.py` is similar to `dev_settings.py` but often uses `os.getenv()` to get values from environment variables. This helps us to keep secrets (for example, passwords, API tokens, and so on) out of version control and decouples settings from particular servers. We'll touch on this again in the *Factor 3 – config* section:

```
from config.common_settings import *
DEBUG = False
assert SECRET_KEY is not None, (
    'Please provide DJANGO_SECRET_KEY '
    'environment variable with a value')
ALLOWED_HOSTS += [
    os.getenv('DJANGO_ALLOWED_HOSTS'),
]
```

First, we import the common settings. Out of an abundance of caution, we ensure that the debug mode is off.

Having a `SECRET_KEY` set is vital to our system staying secure. We `assert` to prevent Django from starting up without `SECRET_KEY`. The `common_settings.py` file should have already set it from an environment variable.

A production website will be accessed on a domain other than `localhost`. We will tell Django what other domains we're serving by appending the `DJANGO_ALLOWED_HOSTS` environment variable to the `ALLOWED_HOSTS` list.

Next, let's update the database configuration:

```
DATABASES['default'].update({
    'NAME': os.getenv('DJANGO_DB_NAME'),
    'USER': os.getenv('DJANGO_DB_USER'),
    'PASSWORD': os.getenv('DJANGO_DB_PASSWORD'),
    'HOST': os.getenv('DJANGO_DB_HOST'),
    'PORT': os.getenv('DJANGO_DB_PORT'),
})
```

We updated the database configuration using values from environment variables.

Now that we have our settings sorted, let's prepare our server.

Preparing our server

Now that our code is ready to go into production, let's prepare our server. In this chapter, we will use Ubuntu 18.04 (Bionic Beaver). If you're running another distribution, then some package names may be different, but the steps we'll take will be the same.

To prepare our server, we will perform the following steps:

1. Installing the required operating system packages
2. Setting up Elasticsearch
3. Creating the database

Let's start by installing the packages we need.

Installing required packages

To run Answerly on our server, we will need to ensure that the correct software is running.

Let's create a list of packages we will need in `ubuntu/packages.txt`:

```
python3
python3-pip
virtualenv

apache2
libapache2-mod-wsgi-py3

postgresql
postgresql-client

openjdk-8-jre-headless
```

The preceding code will install packages for the following:

- Full Python 3 support
- The Apache HTTP Server
- mod_wsgi, the Apache HTTP module for running Python web apps
- The PostgreSQL database server and client
- Java 8, required for Elasticsearch

To install the packages, run the following command:

```
$ sudo apt install -y $(cat /answerly/ubuntu/packages.txt)
```

Next, we'll install our Python packages to a virtual environment:

```
$ mkvirutalenv /opt/answerly.venv
$ source /opt/answerly.venv/bin/activate
$ pip install -r /answerly/requirements.production.txt
```

Great! Now that we have all the packages, we will need to set up Elasticsearch. Unfortunately, Ubuntu doesn't ship with a recent version of Elasticsearch, so we'll install it directly from Elastic instead.

Configuring Elasticsearch

We will get Elasticsearch directly from Elastic. Elastic makes this simple by running a server with Ubuntu-compatible `.deb` packages that we can add to our server (Elastic also ships and supports RPMs, if that's more convenient for you). Finally, we have to remember to rebind Elasticsearch to localhost or we will have an unsecured server running on an open public port.

Installing Elasticsearch

Let's add Elasticsearch to our list of trusted repositories by running the following three commands:

```
$ wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-
key add -
$ sudo apt install apt-transport-https
$ echo "deb https://artifacts.elastic.co/packages/6.x/apt stable main" |
sudo tee -a /etc/apt/sources.list.d/elastic-6.x.list
$ sudo apt update
```

The preceding commands perform the following four steps:

1. Add the Elastic GPG key to the list of trusted GPG keys
2. Ensure that `apt` gets packages over `HTTPS` by installing the `apt-transport-https` package
3. Add a new sources file that lists the Elastic package server so that `apt` knows how to get the Elasticsearch package from Elastic
4. Update the list of available packages (which will now include Elasticsearch)

Now that we have the Elasticsearch available, let's install it:

```
$ sudo apt install elasticsearch
```

Next, let's configure Elasticsearch.

Running Elasticsearch

By default, Elasticsearch is configured to bind to a public IP address and includes no authentication.

To change the address Elasticsearch is running on, let's edit `/etc/elasticsearch/elasticsearch.yml`. Find the line with `network.host` and update it, as follows:

```
network.host: 127.0.0.1
```

If you don't change the `network.host` setting, then you'll be running Elasticsearch with no authentication and on a public IP. Your server getting hacked becomes inevitable.

Finally, we want to make sure that Ubuntu starts Elasticsearch and keeps it running. To accomplish that, we need to tell systemd to start Elasticsearch:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable elasticsearch.service
$ sudo systemctl start elasticsearch.service
```

The preceding commands perform the following three steps:

1. Fully reload systemd, which will then become aware of the newly installed Elasticsearch service
2. Enable the Elasticsearch service so that it starts when the server boots (in case of reboots or shutdown)
3. Start Elasticsearch

If you need to stop the Elasticsearch service, you can use `systemctl`: `sudo systemctl stop elasticsearch.service`.

Now that we have Elasticsearch running, let's configure the database.

Creating the database

Django has support for migrations but cannot create the database or database user by itself. We'll write a script to do this for us now.

Let's add the database creation script to our project in `postgres/make_database.sh`:

```
#!/usr/bin/env bash

psql -v ON_ERROR_STOP=1 <<-EOSQL
    CREATE DATABASE $DJANGO_DB_NAME;
    CREATE USER $DJANGO_DB_USER;
    GRANT ALL ON DATABASE $DJANGO_DB_NAME to "$DJANGO_DB_USER";
    ALTER USER $DJANGO_DB_USER PASSWORD '$DJANGO_DB_PASSWORD';
    ALTER USER $DJANGO_DB_USER CREATEDB;
EOSQL
```

To create the database, let's run the following commands:

```
$ sudo su postgres
$ export DJANGO_DB_NAME=answerly
$ export DJANGO_DB_USER=answerly
$ export DJANGO_DB_PASSWORD=password
$ bash /answerly/postgres/make_database.sh
```

The preceding commands do the following three things:

1. Switch us to be the `postgres` user, who is trusted to connect to the Postgres database without any additional credentials.
2. Set environment variables, describing our new database user and schema. **Remember to change the `password` value to a strong password.**
3. Execute the `make_database.sh` script.

Now that we have our server configured, let's deploy Answerly using Apache and mod_wsgi.

Deploying Answerly with Apache

We will deploy Answerly using Apache and mod_wsgi. mod_wsgi is an open source Apache module that lets Apache host Python programs that implement the **Web Server Gateway Interface** (**WSGI**) specification.

The Apache web server is one of the many great options for deploying Django projects. Many organizations have an operations team who deploy Apache servers, so using Apache can remove some organizational hurdles in using Django for a project. Apache (with mod_wsgi) also knows how to run multiple web apps and route requests between them, unlike our previous configuration in `Chapter 5`, *Deploying with Docker*, where we needed a reverse proxy (NGINX) and web server (uWSGI). The downside of using Apache is that it uses more memory than uWSGI. Also, Apache doesn't have a way of passing environment variables to our WSGI process. On the whole, deploying with Apache can be a really useful and important tool in a Django developer's belt.

To deploy, we will do the following things:

1. Create a virtual host config
2. Update `wsgi.py`
3. Create an environment config file
4. Collect the static files
5. Migrate the database
6. Enable the virtual host

Let's start creating a virtual host config for our Apache web server.

Creating the virtual host config

A single Apache web server can host many websites using different technologies from different locations. To keep each website separate, Apache provides the capacity to define a virtual host. Each virtual host is a logically separate site that serves one or more domains and ports.

Since Apache has already been a great web server, we will use it to serve our static files. The web server serving the static files and our mod_wsgi process won't be competing, because they will run as separate processes, thanks to mod_wsgi's daemon mode. mod_wsgi daemon mode means that Answerly will run in separate processes from the rest of Apache. Apache will still be responsible for starting/stopping these processes.

Let's add the Apache virtual host config to our project under `apache/answerly.apache.conf`:

```
<VirtualHost *:80>

    WSGIDaemonProcess answerly \
        python-home=/opt/answerly.venv \
        python-path=/answerly/django \
        processes=2 \
        threads=15
    WSGIProcessGroup answerly
    WSGIScriptAlias / /answerly/django/config/wsgi.py

    <Directory /answerly/django/config>
        <Files wsgi.py>
            Require all granted
        </Files>
    </Directory>

    Alias /static/ /answerly/django/static_root
    <Directory /answerly/django/static_root>
        Require all granted
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

</VirtualHost>
```

Let's look at the some of these instructions more closely:

- `<VirtualHost *:80>`: This instructs Apache that everything until the closing `</VirtualHost>` tag is part of the virtual host definition.
- `WSGIDaemonProcess`: This configures mod_wsgi to run in daemon mode. The daemon process will be named `answerly`. The `python-home` option defines the virtual environment for the Python process that the daemon will use. The `python-path` option lets us add our modules to the daemon's python so that they can be imported. The `processes` and `threads` options tell Apache how many of each to maintain.
- `WSGIProcessGroup`: This associates this virtual host with the Answerly mod_wsgi daemon. Remember that you keep the `WSGIDaemonProcess` name and the `WSGIProcessGroup` name the same.
- `WSGIScriptAlias`: This describes which requests should be routed to which WSGI script. In our case, all requests should go to Answerly's WSGI script.

- `<Directory /answerly/django/config>`: This block gives all users permission to access our WSGI script.
- `Alias /static/ /answerly/django/static_root`: This routes any request that begins with `/static/` not to mod_wsgi but to our static file root.
- `<Directory /answerly/django/static_root>`: This block gives users permission to access files in `static_root`.
- `ErrorLog` and `CustomLog`: They describe where Apache should send its logs for this virtual host. In our case, we want to log it in the Apache `log` directory (commonly, `/var/log/apache`).

We have now configured Apache to run Answerly. However, if you compare your Apache configuration and your uWSGI configuration from `Chapter 5`, *Deploying with Docker*, you'll notice a difference. In the uWSGI configuration, we provided the environment variables that our `production_settings.py` relies on. However, mod_wsgi doesn't offer us such a facility. Instead, we will update `django/config/wsgi.py` to provide the environment variables that `production_settings.py` needs.

Updating wsgi.py to set environment variables

Now, we will update `django/config/wsgi.py` to provide the environment variables that `production_settings.py` wants but mod_wsgi can't provide. We will also update `wsgi.py` to read a configuration file on startup and then set the environment variables itself. This way, our production settings aren't coupled to mod_wsgi or a config file.

Let's update `django/config/wsgi.py`:

```
import os
import configparser
from django.core.wsgi import get_wsgi_application

if not os.environ.get('DJANGO_SETTINGS_MODULE'):
    parser = configparser.ConfigParser()
    parser.read('/etc/answerly/answerly.ini')
    for name, val in parser['mod_wsgi'].items():
        os.environ[name.upper()] = val

application = get_wsgi_application()
```

In the updated `wsgi.py`, we check whether there is a `DJANGO_SETTINGS_MODULE` environment variable. If it is absent, we parse our config file and set environment variables. Our `for` loop transforms the names of the variables to ensure that they are uppercase since `ConfigParser` makes them `lowercase` by default.

Next, let's create our environment config file.

Creating the environment config file

We'll store our environment config under `/etc/answerly/answerly.ini`. We don't want it stored under `/answerly` because it's not part of our code. This file describes the settings for *just* this server. We should never check this file into version control.

Let's create `/etc/answerly/answerly.ini` on our server:

```
[mod_wsgi]
DJANGO_ALLOWED_HOSTS=localhost
DJANGO_DB_NAME=answerly
DJANGO_DB_USER=answerly
DJANGO_DB_PASSWORD=password
DJANGO_DB_HOST=localhost
DJANGO_DB_PORT=5432
DJANGO_ES_INDEX=answerly
DJANGO_ES_HOST=localhost
DJANGO_ES_PORT=9200
DJANGO_LOG_FILE=/var/log/answerly/answerly.log
DJANGO_SECRET_KEY=a large random value
DJANGO_SETTINGS_MODULE=config.production_settings
```

The following are the two things to remember about this file:

- Remember to set `DJANGO_DB_PASSWORD` to the same value you set in when you ran the `make_database.sh` script. *Remember to make sure that this password is strong and secret*.
- Remember to set a strong `DJANGO_SECRET_KEY` value.

We should now have our environment set up for Apache. Next, let's migrate the database.

Migrating the database

We created the database for Answerly in a previous step, but we didn't create the tables. Let's now migrate the database using Django's built-in migration tools.

On the server, we want to execute the following commands:

```
$ cd /answerly/django
$ source /opt/answerly.venv/bin/activate
$ export DJANGO_SECRET_KEY=anything
$ export DJANGO_DB_HOST=127.0.0.1
$ export DJANGO_DB_PORT=5432
$ export DJANGO_LOG_FILE=/var/log/answerly/answerly.log
$ export DJANGO_DB_USER=myqa
$ export DJANGO_DB_NAME=myqa
$ export DJANGO_DB_PASSWORD=password
$ sudo python3 manage.py migrate --settings=config.production_settings
```

Our `django/config/production_settings.py` will require us to provide `DJANGO_SECRET_KEY` with a value, but it won't be used in this case. However, providing the correct value for `DJANGO_DB_PASSWORD` and the other `DJANGO_DB` variables is critical.

Once our `migrate` command returns successful, then our database will have all the tables we need.

Next, let's make our static (JavaScript/CSS/image) files available to our users.

Collecting static files

In our virtual host config, we configured Apache to serve our static (JS, CSS, image, and so on) files. For Apache to serve these files, we need to collect them all under one parent directory. Let's use Django's built-in `manage.py collectstatic` command to do just that.

On the server, let's run the following commands:

```
$ cd /answerly/django
$ source /opt/answerly.venv/bin/activate
$ export DJANGO_SECRET_KEY=anything
$ export DJANGO_LOG_FILE=/var/log/answerly/answerly.log
$ sudo python3 manage.py collectstatic --
settings=config.production_settings --no-input
```

The preceding commands will copy static files from all our installed apps into `/answerly/django/static_root` (per our `STATIC_ROOT` definition in `production_settings.py`). Our virtual host config tells Apache to serve these files directly.

Now, let's tell Apache to start serving Answerly.

Enabling the Answerly virtual host

To have Apache serve Answerly to users, we will need to enable the virtual host config we created the preceding section, creating the virtual host config. To enable a virtual host in Apache, we will add a soft link point at the virtual host config to Apache's `site-enabled` directory and tell Apache to reload its configuration.

First, let's add our soft link to the Apache `site-enabled` directory:

```
$ sudo ln -s /answerly/apache/answerly.apache.conf /etc/apache/site-
enabled/000-answerly.conf
```

We prefix our softlink with `001` to control what our config gets loaded. Apache loads site configs by filename in character ordinal order (for example, `B` comes before `a` in Unicode/ASCII encoding). The prefix is used to make the order more obvious.

Apache is frequently packaged with a default site. Check out `/etc/apache/sites-enabled/` for sites you don't want to run. Since everything in there should be a soft link, they should be safe to delete.

To activate the virtual host, we will need to reload Apache's configuration:

```
$ sudo systemctl reload  apache2.service
```

Congratulations! You've deployed Answerly on your server.

A quick review of the section

In this chapter so far, we've looked at how to deploy Django with Apache and mod_wsgi. First, we configured our server by installing packages from Ubuntu and Elastic (for Elasticsearch). Then, we configured Apache to run Answerly as a virtual host. Our Django code will be executed by mod_wsgi.

At this point, we've seen two very different deployment, one using Docker and one using Apache and mod_wsgi. Despite being very different environments, we've followed many similar practices. Let's look at how Django best practices come out of the popular twelve-factor app methodology.

Deploying Django projects as twelve-factor apps

The *twelve-factor app* document explains a methodology to develop web apps and services. These principles were documented in 2011 by Adam Wiggins and others primarily on their experience at Heroku (a popular Platform as a Service, PaaS, provider). Heroku was one of the first PaaS that helped developers build easy-to-scale web applications and services. Since being posted, the principles of twelve-factor apps have shaped a lot of the thinking about how to build and deploy SaaS apps—like web apps.

The twelve-factors provide many benefits, as follows:

- Easing automation and onboarding using declarative formats
- Emphasizing portability across deployed environments
- Encouraging production/development environment parity and continuous deployment and integration
- Simplifying scaling without requiring re-architecting

However, when evaluating the twelve factors, it's important to remember that they are strongly coupled to Heroku's approach to deployment. Not all platforms (or PaaS providers) have exactly the same approach. This doesn't make the twelve factors right and other approaches wrong, nor vice versa. Rather the twelve factors are useful principles to keep in mind. You should adapt them to help your projects, just as you would with any methodology.

The twelve factor use of the word *app* is different to Django's usability:

- A Django project is the equivalent of a twelve factor app
- A Django app is the equivalent of a twelve factor library

In this section, we will examine what each of the twelve-factors means and how they can be applied to your Django projects.

Factor 1 – Code base

> "One codebase tracked in revision control, many deploys" – `12factor.net`

This factor emphasizes the following two things:

- All code should be tracked in a version-controlled code repository (repo)
- Each deployment should be able to reference a single version/commit in that repo

This means that when we experience a bug, we know exactly the version of the code that is responsible for that. If our project spans multiple repos, the twelve-factor approach requires that shared code be refactored into libraries and tracked as dependencies (refer to the *Factor 2 – Dependencies* section). If multiple projects use the same repository, then they should be refactored into separate repositories (sometimes called *multi repo*). Over the years since twelve-factor was first published, multirepo versus monorepo (where a single repo is used for multiple projects) has become increasingly debated. Some large projects have found benefits to using a mono repo. Other projects have found success with multiple repos.

Fundamentally, this factor strives to ensure that we know what is running in which environment.

We can write our Django apps in a reusable way so that they can be hosted as libraries that are installed with `pip` (multirepo style). Alternatively, you can host all your Django projects and apps in the same repo (monorepo) by modifying the Python path of your Django project.

Factor 2 – Dependencies

> "Explicitly declare and isolate dependencies" – `12 factor.net`

A twelve-factor app shouldn't assume anything about its environment. The libraries and tools a project uses must be declared by the project and installed as part of the deployment (refer to *Factor 5 – Build, release, and run* section). All running twelve-factor apps should be isolated from each other.

Django projects benefit from Python's rich toolset. “In Python there are two separate tools for these steps – Pip is used for declaration and Virtualenv for isolation” (`https://12factor.net/dependencies`). In Answerly, we also used a list of Ubuntu packages that we installed with `apt`.

Factor 3 – Config

> "Store config in the environment" – `12factor.net`

The twelve-factor app methodology provides a useful definition of a config:

> "An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc)" – `https://12factor.net/config`

The twelve-factor app methodology also encourages the use of environment variables for communicating config values to our code. This means that if there's a problem, we can test exactly the code that was deployed (provided by Factor 1) with the exact config used. We can also check whether an error is a config issue or a code issue by deploying the same code with a different config.

In Django, our config is referenced by our `settings.py` files. In both MyMDB and Answerly, we've seen common config values such as `SECRET_KEY`, database credentials, and API keys (for example, AWS keys) passed by environment variables.

However, this is an area where Django best practices differ from the strictest reading of a twelve-factor app. Django projects generally create a separate settings file for staging, production, and local development with most settings hardcoded. It's primarily credentials and secrets which are passed as environment variables.

Factor 4 – Backing services

> "Treat backing services as attached resources" – `12factor.net`

A twelve-factor app should not care where a backing service (for example, database) is located and should always access it via a URL. The benefit of this is that our code is not coupled to a particular environment. This approach also permits each piece of our architecture to scale independently.

Answerly, as deployed in this chapter, is located on the same server as its database. However, we don't use a local authentication mechanism but instead provide Django with a host, port, and credentials. This way, we could move our database to another server and no code would have to be changed. We would simply update our config.

Django is written with the assumption that we will treat most services as attached resources (for example, most database documentation assumes this). We still need to practice this principle when working with third-party libraries.

Factor 5 – Build, release, and run

> "Strictly separate build and run stages" – `12factor.net`

The twelve-factor approach encourages a deployment to be divided into three distinct steps:

1. **Build**: Where the code and dependencies are gathered into a single bundle (a *build*)
2. **Release**: Where the build is combined with a config and ready for execution
3. **Run**: Where the combined build and config are executed

A twelve-factor app further requires each release to have a unique ID so that it can be identified.

This level of deployment detail is beyond Django's scope, and there's a variety of levels of adherence to this strict three-step model. A project that uses Django and Docker, as seen in `Chapter 5`, *Deploying with Docker*, may adhere to it very closely. MyMDB had a clear build with all the dependencies bundled in the Docker image. However, in this chapter, we never made a bundled build. Instead, we installed dependencies (running `pip install`) after our code was already on our server. Many projects succeed with this simple model. However, as the project scales, this may cause complications. Answerly's deployment shows how twelve-factor principles may be bent and still work for some projects.

Factor 6 – Processes

> "Execute the app as one or more stateless processes" – `12factor.net`

The focus of this factor is that app processes should be *stateless*. Each task is executed without relying on a previous task having left data behind. Instead, state should be stored in backing services (refer to *Factor 4 – Backing services* section), such as a database or external cache. This enables an app to scale easily, because all processes are equally eligible to process a request.

Django is built around this assumption. Even sessions, where a user's login state is stored, isn't saved in a process but in the database by default. Instances of view classes are never reused. The only place where Django comes close to violating this is one of the cache backends (local memory cache). However, as we discussed, that's an inefficient backend. Generally, Django projects use a backing service (for example, memcached) for their caches.

Factor 7 – Port binding

> "Export services via port binding" – `12factor.net`

The focus of this factor is that our process should be accessed directly through its port. Accessing a project should be a matter of sending a properly formed request to `app.example.com:1234`. Further, a twelve-factor app should not be run as an Apache module or web server container. If our project needs to parse HTTP requests, it should use library (refer to *Factor 2 – Dependencies* section) to parse them.

Django follows parts of this principle. Users access a Django project over an HTTP port using HTTP. One aspect of Django that diverges from twelve-factors is that it's almost always run as a child process of a web server (whether Apache, uWSGI, or something else). It's the web server, not Django, that performs the port binding. However, this minor difference has not kept Django projects from scaling effectively.

Factor 8 – Concurrency

> "Scale out via the process model" – `12factor.net`

The twelve-factor app principles are focused on scaling (a vital concern for a PaaS provider like Heroku). In factor 8, we saw how the trade-offs and decisions made previously come together to help a project scale.

Since a project runs as a stateless process (refer to *Factor 6 – Processes* section) available as a port (refer to *Factor 7 – Port binding* section), concurrency is just a matter of having more processes (across one or more machines). The processes don't need to care whether they're on the same machine or not since any state (like a question's answer) is stored in a backing service (refer to *Factor 4 – Backing services* section) such as a database. Factor 8 tells us to trust the Unix process model for running services instead of daemonizing or creating PID files.

Since Django projects run as child processes of a web server, they often adapt this principle. Django projects that need to scale often use a combination of reverse proxy (for example, Nginx) and lightweight web server (for example, uWSGI or Gunicorn). Django projects don't directly concern themselves with how processes are managed, but follow the best practice for the web server they're using.

Factor 9 – Disposability

> "Maximize robustness with fast startup and graceful shutdown" – `12factor.net`

The disposibility factor has two parts. Firstly, a twelve-factor app should be able to start processing requests on its port soon after the process starts. Remember that all its dependencies (refer to *Factor 2 – Dependencies* section) have already been installed (refer to *Factor 5 – Build, release, and run* section). A twelve-factor app should handle a process stopping or shutting gracefully. The process shouldn't put a twelve-factor app into an invalid state.

Django projects are able to shut down gracefully because Django wraps each request in an atomic transaction by default. If a Django process (whether managed by uWSGI, Apache, or anything else) is stopped while a request is only partially processed, the transaction will never be committed. The database will discard the transaction. When we're dealing with other backing services (for example, S3 or Elasticsearch) that don't support transactions, we have to make sure that we consider this in our design.

Factor 10 – Dev/prod parity

> "Keep development, staging, and production as similar as possible"
> – `12factor.net`

All environments that a twelve-factor app run in should be as similar as possible. This is much easier when a twelve-factor app is a simple process (refer to *Factor 6 – Processes* section). This also includes the backing services the twelve-factor app uses (refer to *Factor 4 – Backing services* section). For example, a twelve-factor app's development environment should include the same database as the production environment. Tools such as Docker and Vagrant can make this much easier to accomplish today.

The general Django best practice is to use the same database (and other backing services) in development and production. In this book, we've striven to do so. However, the Django community often uses the `manage.py runserver` command in development, as opposed to running uWSGI or Apache.

Factor 11 – Logs

> "Treat logs as event streams" – `12factor.net`

Logs should be just output as an unbuffered `stdout` stream, and a *twelve-factor app never concerns itself with routing or storage of its output stream* (`https://12factor.net/logs`). When the process runs, it should just output unbuffered content to `stdout`. Whoever starts the process (whether a developer or a production server's init process) can then redirect that stream appropriately.

A Django project generally uses Python's logging module. This can support writing to a log file or outputting an unbuffered stream. Generally, Django projects append to a file. That file may be processed or rotated separately (for example, using the `logrotate` utility).

Factor 12 – Admin processes

> "Run admin/management tasks as one-off processes" – `12factor.net`

All projects require a one-off task to be run from time to time (for example, database migration). When a twelve-factor app's one-off task is run, it should be run as a separate process from the processes that handle regular requests. However, the one-off process should run with the same environment as all other processes.

In Django that means using the same virtual environment, settings file, and environment variables for running our `manage.py` tasks as our normal process. This is what we did earlier when we migrated the database.

A quick review of the section

After reviewing all the principles of a twelve-factor app, we will take a look at how Django projects are able to follow these principles to help make our project easy to deploy, scale, and automate.

The main difference between a Django project and a strict twelve-factor app is that Django apps are run by a web server rather than as separate processes (Factor 6). However, as long as we avoid complicated web server configurations (as we do in this book), we can continue to gain the benefits of being a twelve-factor app.

Summary

In this chapter, we focused on deploying Django to a Linux server running Apache and mod_wsgi. We've also reviewed the principles of a twelve factor app and how a Django app can use them to be easy to deploy, scale, and automate.

Congratulations! You've launched Answerly.

In the next chapter, we'll look at creating a mailing list management app called MailApe.

10
Starting Mail Ape

In this chapter, we'll begin building Mail Ape, a mailing list manager that will let users start mailing lists, sign up for mailing lists, and then message people. Subscribers will have to confirm their subscription to a mailing list and be able to unsubscribe. This will help us to ensure that Mail Ape isn't used to serve spam to users.

In this chapter, we will build the core Django functionality of Mail Ape:

- We'll build models that describe Mail Ape, including `MailingList` and `Subscriber`
- We'll use Django's Class-Based Views to create web pages
- We'll use Django's built-in authentication functionality to let users log in
- We'll make sure that only the owner of a `MailingList` model instance can email its subscribers
- We'll create templates to generate the HTML to display the forms to subscribe and email to our users
- We'll run Mail Ape locally using Django's built-in development server

The code for this project is available online at `https://github.com/tomaratyn/MailApe`.

Django follows the **Model View Template** (**MVT**) pattern to separate model, control, and presentation logic and encourage reusability. Models represent the data we'll store in the database. Views are responsible for handling a request and returning a response. Views should not have HTML. Templates are responsible for the body of a response and defining the HTML. This separation of responsibilities has proven to make it easy to write code.

Let's start by creating the Mail Ape project.

Creating the Mail Ape project

In this section, we will create the MailApe project:

```
$ mkdir mailape
$ cd mailape
```

All the paths in this part of the book will be relative to this directory.

Listing our Python dependencies

Next, let's create a `requirements.txt` file to track our Python dependencies:

```
django<2.1
psycopg2<2.8
django-markdownify==0.3.0
django-crispy-forms==1.7.0
```

Now that we know our requirements, we can install them, as follows:

```
$ pip install -r requirements.txt
```

This will install the following four libraries:

- `Django`: Our favorite web app framework
- `psycopg2`: The Python PostgreSQL library; we'll use PostgreSQL in both production and development
- `django-markdownify`: A library that makes it easy to render markdown in a Django template
- `django-crsipy-forms`: A library that makes it easy to create Django forms in templates

With Django installed, we can use the `django-admin` utility to create our project.

Creating our Django project and apps

A Django project is composed of a configuration directory and one or more Django apps. The actual functionality of a project is encapsulated by the installed apps. By default, the configuration directory is named after the project.

A web app is often composed of much more than just the Django code that is executed. We need configuration files, system dependencies, and documentation. To help future developers (including our future selves), we will strive to label each directory clearly:

```
$ django-admin startporject config
$ mv config django
$ tree django
django
├── config
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

With this approach, our directory structure is clear about the location of our Django code and configuration.

Next, let's create the apps that will encapsulate our functionality:

```
$ python manage.py startapp mailinglist
$ python manage.py startapp user
```

For each app, we should create a URLConf. A URLConf ensures that requests get routed to the right view. A URLConf is a list of paths, the view that serves the path, and the name for the path. One great thing about URLConfs is that they can include each other. When a Django project is created, it gets a root URLConf (ours is at `django/config/urls.py`). Since a URLConf may include other URLConfs, the name provides a vital way to reference a URL path to a view without knowing the full URL path to the view.

Creating our app's URLConfs

Let's create a URLConf for the `mailinglist` app in `django/mailinglist/urls.py`:

```
from django.urls import path

from mailinglist import views

app_name = 'mailinglist'

urlpatterns = [
]
```

The `app_name` variable is used to scope the paths in case of name collisions. When resolving a pathname, we can prefix it with `mailinglist:` to ensure that it's from this app. As we build our views, we'll add `path` s to the `urlpatterns` list.

Next, let's create another URLConf in the `user` app by creating `django/user/urls.py`:

```
from django.contrib.auth.views import LoginView, LogoutView
from django.urls import path

import user.views

app_name = 'user'
urlpatterns = [
]
```

Great! Now, let's include them in the root ULRConf that's located in `django/config/urls.py`:

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

import mailinglist.urls
import user.urls

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

The root URLConf is just like our app's URLConfs. It has a list of `path()` objects. The `path()` objects in the root URLConfs usually don't have views but `include()` other URLConfs. Let's take a look at the two new functions here:

- `path()`: This takes a string and either a view or the result of `include()`. Django will iterate over the `path()`s in a URLConf until it finds one that matches the path of a request. Django will then pass the request to that view or URLConf. If it's a URLConf, then that list of `path()`s is checked.
- `include()`: This takes a URLConf and a namespace name. A namespace isolates a URLConfs from each other so that we can prevent name collisions, ensuring that we can differentiate `appA:index` from `appB:index`. `include()` returns a tuple; the object at `admin.site.urls` has been already a correctly formatted tuple, so we don't have to use `include()`. Generally, we always use `include()`.

If Django can't find a `path()` object that matches a request's path, then it will return a 404 response.

The result of this URLConf is as follows:

- Any request starting with `admin/` will be routed to the admin app's URLConf
- Any request starting with `mailinglist/` will be routed to the `mailinglist` app's URLConf
- Any request starting with `user/` will be routed to the `user` app's URLConf

Installing our project's apps

Let's update `django/config/settings.py` to install our apps. We'll change the `INSTALLED_APPS` setting as shown in the following code snippet:

```
INSTALLED_APPS = [
    'user',
    'mailinglist',

    'crispy_forms',
    'markdownify',

    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Now that we have our project and apps configured, let's create models for our `mailinglist` app.

Creating the mailinglist models

In this section, we'll create the models for our `mailinglist` app. Django provides a rich and powerful ORM that will let us define our models in Python without having to deal with the database directly. The ORM converts our Django classes, fields, and objects into relational database concepts:

- A model class maps to a relational database table
- A field maps to a relational database column
- A model instance maps to a relational database row

Each model also comes with a default manager available in the `objects` attribute. A manager provides a starting point for running queries on a model. One of the most important methods a manager has is `create()`. We can use `create()` to create an instance of the model in our database. A manager is also the starting point to get a `QuerySet` for our model.

A `QuerySet` represents a database query for models. `QuerySet`s are lazy and only execute when they're iterated or converted to a `bool`. A `QuerySet` API offers most the functionality of SQL without being tied a particular database. Two particularly useful methods are `QuerySet.filter()` and `QuerySet.exclude()`. `QuerySet.filter()` lets us filter the results of the `QuerySet` to only those matching the provided criteria. `QuerySet.exclude()` lets us exclude results that don't match the criteria.

Let's start with the first model, `MailingList`.

Creating the MailingList model

Our `MailingList` model will represent a mailing list that one of our users has created. This will be an important model for our system because many other models will be referring to it. We can also anticipate that the `id` of a `MailingList` will have to be publicly exposed in order to relate subscribers back to it. To avoid letting users enumerate all the mailing lists in Mail Ape, we want to make sure that our `MailingList` IDs are nonsequential.

Let's add our `MailingList` model to `django/mailinglist/models.py`:

```
import uuid

from django.conf import settings
from django.db import models
from django.urls import reverse


class MailingList(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    name = models.CharField(max_length=140)
    owner = models.ForeignKey(to=settings.AUTH_USER_MODEL,
                              on_delete=models.CASCADE)

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse(
            'mailinglist:manage_mailinglist',
            kwargs={'pk': self.id}
        )

    def user_can_use_mailing_list(self, user):
        return user == self.owner
```

Let's take a closer look at our `MailingList` model:

- `class MailingList(models.Model)`:: All Django models must inherit from the `Model` class.
- `id = models.UUIDField`: This is the first time we've specified the `id` field for a model. Usually, we let Django provide one for us automatically. In this case, we wanted nonsequential IDs, so we used a field that provides **Universally Unique Identifiers** (**UUIDs**). Django will create the proper database field when we generate the migrations (refer to the *Creating database migrations* section). However, we have to generate the UUID in Python. To generate new UUIDs for each new model, we used the `default` argument and Python's `uuid4` function. To tell Django that our `id` field is the primary key, we used the `primary_key` argument. We further passed `editable=False` to prevent changes to the `id` attribute.

- `name = models.CharField`: This will represent the mailing list's name. A `CharField` will get converted to a `VARCHAR` column, so we must provide it with a `max_length` argument.
- `owner = models.ForeignKey`: This is a foreign key to Django's user model. In our case, we will use the default `django.contrib.auth.models.User` class. We follow the Django best practice of avoiding hardcoding this model. By referencing `settings.AUTH_USER_MODEL`, we don't couple our app to the project too tightly. This encourages future reuse. The `on_delete=models.CASCADE` argument means that if a user is deleted, all their `MailingList` model instances will be deleted too.
- `def __str__(self)`: This defines how to convert a mailing list to a `str`. Both Django and Python will use this when a `MailingList` needs to be printed out or displayed.
- `def get_absolute_url(self)`: This is a common method on Django models. `get_absolute_url()` returns a URL path that represents the model. In our case, we return the management page for this mailing list. We don't hardcode the path. Instead, we use `reverse()` to resolve the path at runtime by providing the name of the URL. We'll look at named URLs in the *Creating the URLConf* section.
- `def user_can_use_mailing_list(self, user)`: This is a method we've added for our own convenience. It checks whether a user can use (meaning view-related items and/or send messages) to this mailing list. Django's *Fat models* philosophy encourages placing code for decisions like this in models rather than in views. This gives us a central place for decisions, ensuring that you **Don't Repeat Yourself** (**DRY**).

We now have our `MailingList` model. Next, let's create a model to capture the mailing list's subscribers.

Creating the Subscriber model

In this section, we will create a `Subscriber` model. A `Subscriber` model can only belong to one `MailingList` and must confirm their subscription. Since we'll need to reference a subscriber for their confirm and unsubscribe pages, we'll want their `id` instance to also be nonsequential.

Let's create the `Subscriber` model in `django/mailinglist/models.py`:

```
class Subscriber(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    email = models.EmailField()
    confirmed = models.BooleanField(default=False)
    mailing_list = models.ForeignKey(to=MailingList,
on_delete=models.CASCADE)

    class Meta:
        unique_together = ['email', 'mailing_list', ]
```

The `Subscriber` model has some similarities to the `MailingList` model. The base class and `UUIDField` function the same. Let's take a look at some of the differences:

- `models.EmailField()`: This is a specialized `CharField` but does extra validation to ensure that the value is a valid email address.
- `models.BooleanField(default=False)`: This lets us store `True/False` values. We need to use to this to track whether a user really intends to subscribe to a mailing list.
- `models.ForeignKey(to=MailingList...)`: This lets us create a foreign key between `Subscriber` and `MailingList` model instances.
- `unique_together`: This is an attribute of the `Meta` inner class of `Subscriber`. A `Meta` inner class lets us specify information on the table. For example, `unique_together` lets us add an additional unique constraint on a table. In this case, we prevent a user from signing up twice with the same email.

Now that we can track `Subscriber` model instances, let's track the messages our users want to send to their `MailingList`.

Creating the Message model

Our users will want to send messages to their `Subscriber` model instances of `MailingList`. In order to know what to send to these subscribers, we will need to store the messages as a Django model.

A `Message` should belong to a `MailingList` and have a nonsequential `id`. We need to save the subject and body of these messages. We will also want to track when the sending began and completed.

Let's add the `Message` model to `django/mailinglist/models.py`:

```
class Message(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    mailing_list = models.ForeignKey(to=MailingList,
on_delete=models.CASCADE)
    subject = models.CharField(max_length=140)
    body = models.TextField()
    started = models.DateTimeField(default=None, null=True)
    finished = models.DateTimeField(default=None, null=True)
```

Again, the `Message` model is very similar to our preceding models in its base class and fields. We do see some new fields in this model. Let's take a closer look at these new fields:

- `models.TextField()`: This is used to store arbitrarily long character data. All major databases have a `TEXT` column type. This is useful to store the `body` attribute of our user's `Message`.
- `models.DateTimeField(default=None, null=True)`: This is used to store date and time values. In Postgres, this becomes a `TIMESTAMP` column. The `null` argument tells Django that this column should be able to accept a `NULL` value. By default, all fields have a `NOT NULL` constraint on them.

We now have our models. Let's create them in our database with database migrations.

Using database migrations

Database migrations describe how to get a database to a particular state. In this section, we will do the following things:

- Create a database migration for our `mailinglist` app models
- Run the migration on a Postgres database

When we make a change to our models, we can have Django generate the code for creating those tables, fields, and constraints. The migrations that Django generates are created using an API that is also available to Django developers. If we need to do a complicated migration, we can write a migration ourselves. Remember that a proper migration includes code for both applying and reverting a migration. If there's a problem, we want to have a way to undo our migration. When Django generates a migration, it always generates both migrations for us.

Let's start by configuring Django to connect to our PostgreSQL database.

Configuring the database

To configure Django to connect to our Postgres database, we will need to update the `DATABASES` setting in `django/config/settings.py`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mailape',
        'USER': 'mailape',
        'PASSWORD': 'development',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

You should not hardcode the password to a production database in your `settings.py` file. If you're connecting to a shared or online instance, set the username, password, and host using environment variables and access them using `os.getenv()`, like we did in our previous production deployment chapters (`Chapter 5`, *Deploying with Docker*, and `Chapter 9`, *Deploying Answerly*).

Django cannot create a database and users by itself. We must do that ourselves. You can find a script for doing this in the code for this chapter.

Next, let's create the migrations for models.

Creating database migrations

To create our database migrations, we will use the `manage.py` script that Django put at the top of the Django project (`django/manage.py`):

```
$ cd django
$ python manage.py makemigrations
Migrations for 'mailinglist':
  mailinglist/migrations/0001_initial.py
    - Create model MailingList
    - Create model Message
    - Create model Subscriber
    - Alter unique_together for subscriber (1 constraint(s))
```

Great! Now that we have the migrations, we can run them on our local development database.

Running database migrations

We use `manage.py` to apply our database migrations to a running database. On the command line, execute the following:

```
$ cd django
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, mailinglist, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying 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 mailinglist.0001_initial... OK
  Applying sessions.0001_initial... OK
```

When we run `manage.py migrate` without providing an app, it will run all migrations on all installed Django apps. Our database now has the tables for the `mailinglist` app models and the `auth` app's models (including the `User` model).

Now that we have our models and database set up, let's make sure that we can validate the user's input for these models using Django's forms API.

MailingList forms

One of the common issues that developers have to solve is how to validate a user input. Django provides input validation through its forms API. The forms API can be used to describe an HTML form using an API very similar to the models API. If we want to create a form that describes a Django model, then the Django form's `ModelForm` offers us a shortcut. We only have to describe what we're changing from the default form representation for the model.

When a Django form is instantiated, it can be provided with any of the three following arguments:

- `data`: The raw input that the end users request
- `initial`: The known safe initial values that we may set for a form
- `instance`: The instance the form is describing, only on `ModelForm`

If a form has been provided `data`, then it is called a bound form. Bound forms can validate their `data` by calling `is_valid()`. A validated form's safe-to-use data is available under the `cleaned_data` dictionary (keyed on the field's name). Errors are available via the `errors` property, which returns a dictionary. A bound `ModelForm` can also create or update its model instance with the `save()` method.

Even if none of the arguments are provided, a form is still able to print itself out as HTML, making our templates much simpler. This mechanism helps us achieve the goal of *dumb templates*.

Let's start creating our forms by creating the `SubscriberForm` class.

Creating the Subscriber form

An important task Mail Ape must perform is to accept emails of a new `Subscriber` for a `MailingList`. Let's create a form to do that validation for us.

`SubscriberForm` must be able to validate input as a valid email. We also want it to save our new `Subscriber` model instance and associate it with the proper `MailingList` model instance.

Let's create that form in `django/mailinglist/forms.py`:

```
from django import forms

from mailinglist.models import MailingList, Subscriber


class SubscriberForm(forms.ModelForm):
    mailing_list = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=MailingList.objects.all(),
        disabled=True,
    )

    class Meta:
        model = Subscriber
        fields = ['mailing_list', 'email', ]
```

Let's take a closer look at our `SubscriberForm`:

- `class SubscriberForm(forms.ModelForm)`:: This shows that our form is derived from `ModelForm`. `ModelForm` knows to check our inner `Meta` class for information on the model and fields that can be used as the basis of this form.
- `mailing_list = forms.ModelChoiceField`: This tells our form to use our custom configured `ModelChoiceField` instead of the default that the forms API would use. By default, Django will show a `ModelChoiceField` that would render as a drop-down box. A user could use the dropdown to pick the associated model. In our case, we don't want the user to be able to make that choice. When we show a rendered `SubscriberForm`, we want it be configured for a particular mailing list. To this end, we change the `widget` argument to be a `HiddenInput` class and mark the field as `disabled`. Our form needs to know the `MailingList` model instances that are valid for this form. We provide a `QuerySet` object that matches all `MailingList` model instances.
- `model = Subscriber`: This tells the form's `Meta` inner class that this form is based on the `Subscriber` model.
- `fields = ['mailing_list', 'email', ]`: This tells the form to only include the following fields from the model in the form.

Next, let's make a form for capturing the `Message`s that our users want to send to their `MailingList`.

Creating the Message Form

Our users will want to send `Messages` to their `MailingLists`. We'll provide a web page with a form where users can create these messages. Before we can create the page, let's create the form.

Let's add our `MessageForm` class to `django/mailinglist/forms.py`:

```
from django import forms

from mailinglist.models import MailingList, Message


class MessageForm(forms.ModelForm):
    mailing_list = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=MailingList.objects.all(),
        disabled=True,
    )

    class Meta:
        model = Message
        fields = ['mailing_list', 'subject', 'body', ]
```

As you may have noticed in the preceding code, `MessageForm` works just like `SubscriberFrom`. The only difference is that we've listed a different model and different fields in the `Meta` inner class.

Next, let's create the `MailingListForm` class, which we'll use to accept input for the name of the mailing list.

Creating the MailingList form

Now, we'll create a `MailingListForm`, which will accept the name and owner of a mailing list. We will use the same `HiddenInput` and `disabled` field pattern as before but this time on the `owner` field. We want to make sure that a user can't change the owner of the mailing list.

Let's add our form to `django/mailinglist/forms.py`:

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

from mailinglist.models import MailingList


class MailingListForm(forms.ModelForm):
    owner = forms.ModelChoiceField(
        widget=forms.HiddenInput,
        queryset=get_user_model().objects.all(),
        disabled=True,
    )

    class Meta:
        model = MailingList
        fields = ['owner', 'name']
```

The `MailingListForm` is very similar to our previous forms, but introduces a new function, `get_user_model()`. We need to use `get_user_model()` because we don't want to couple ourselves to a particular user model, but we need access to that model's manager to get a `QuerySet`.

Now that we have our forms, we can create the views for our `mailinglist` Django app.

Creating MailingList views and templates

In the preceding section, we created forms that we can use to collect and validate user input. In this section, we will create the views and templates that actually communicate with the user. A template defines the HTML of a document.

Fundamentally, a Django view is a function that accepts a request and returns a response. While we won't be using these **Function-Based Views** (**FBVs**) in this book, it's important to remember that all a view needs to do is meet those two responsibilities. If processing a view also causes another action to occur (for example, sending an email), then we should put that code in a service module rather than directly in the view.

A lot of the work that web developers face is repetitive (for example, processing a form, showing a particular model, listing all instances of that model, and so on). Django's battery included philosophy means that it includes tools to make these kinds of repetitive tasks easier.

Django makes common web developer tasks easier by offering a rich suite of **class-based views** (**CBVs**). CBVs use the principles of **Object-Oriented Programming** (**OOP**) to increase code reuse. Django comes with a rich suite of CBVs that makes it easy to process a form or show an HTML page for a model instance.

The HTML view returns come from rendering a template. Templates in Django are generally written in Django's template language. Django can also support other template languages (for example, Jinja). Generally, each view is associated with a template.

Let's start by creating some resources many of our views will need.

Common resources

In this section, we will create some common resources that our views and templates will need:

- We'll create a base template, which all our other templates can extend. Using the same base template across all our pages will give Mail Ape a unifying look and feel.
- We'll create a `MailingListOwnerMixin` class, which will let us protect mailing lists messages from unauthorized access.

Let's start by creating a base template.

Creating a base template

Let's create a base template for Mail Ape. This template will be used by all our pages to give our entire web app a consistent look.

The **Django template language** (**DTL**) lets us write our HTML (or other text-based format) and lets us use *tags*, *variables*, and *filters* to execute code to customize the HTML. Let's take a closer look at those three concepts:

- *tags*: They are surrounded by `{% %}` and may (`{% block body%}{% endblock %}`) or may not (`{% url "myurl" %}`) contain a body.

- *variables*: They are surrounded by `{{ }}` and must be set in the template's context (for example, `{{ mailinglist }}`). Though DTL variables are like Python variables, there are differences. The two most critical ones are around executables and dictionaries. Firstly, DTL does not have a syntax to pass arguments to an executable (you never have to use `{{foo(1)}}`). If you reference a variable and it is callable (for example, a function), then the Django template language will call it and return the result (for example, `{{mailinglist.get_absolute_url}}`). Secondly, DTL doesn't distinguish among object attributes, items in a list, and items in a dictionary. All three are accessed using a dot: `{{mailinglist.name}}`, `{{mylist.1}}`, and `{{mydict.mykey}}`.
- *filters*: They follow a variable and modify its value (for example, `{{ mailinglist.name | upper}}` will return the mailing lists' name in uppercase).

We'll take a look at examples of all three as we continue creating Mail Ape.

Let's create a common templates directory—`django/templates`—and put our template in `django/templates/base.html`:

```
<!DOCTYPE html>
<html lang="en" >
<head >
  <meta charset="UTF-8" >
  <title >{% block title %}{% endblock %}</title >
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.3/css/bootstrap.
min.css"
  />
</head >
<body >
<div class="container" >
  <nav class="navbar navbar-light bg-light" >
    <a class="navbar-brand" href="#" >Mail Ape </a >
    <ul class="navbar-nav" >
      <li class="nav-item" >
        <a class="nav-link"
           href="{% url "mailinglist:mailinglist_list" %}" >
          Your Mailing Lists
        </a >
      </li >
      {% if request.user.is_authenticated %}
        <li class="nav-item" >
          <a class="nav-link"
             href="{% url "user:logout" %}" >
            Logout
```

```
            </a >
          </li >
        {% else %}
          <li class="nav-item" >
            <a class="nav-link"
               href="{% url "user:login" %}" >
              Your Mailing Lists
            </a >
          </li >
          <li class="nav-item" >
            <a class="nav-link"
               href="{% url "user:register" %}" >
              Your Mailing Lists
            </a >
          </li >
        {% endif %}
      </ul >
    </nav >
    {% block body %}
    {% endblock %}
  </div >
  </body >
  </html >
```

In our base template, we will note examples of the following three tags:

- `{% url ... %}`: This returns the path to a view. This works just like the `reverse()` function we saw earlier but in a Django template.
- `{% if ... %} ... {% else %} ... {% endif %}`: This works just like a Python developer would expect. The `{% else %}` clause is optional. The Django template language also supports `{% elif ... %}` if we need to choose among multiple choices.
- `{% block ... %}`: This defines a block that a template, which extends `base.html`, can replace with its own content. We have two blocks, `body` and `title`.

We now have a base template that our other templates can use by just providing body and title blocks.

Now that we have our template, we have to tell Django where to find it. Let's update `django/config/settings.py` to let Django know about our new `django/templates` directory.

In `django/config/settings.py`, find the line that starts with `Templates`. We will need to add our `templates` directory to the list under the `DIRS` key:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR, 'templates'),
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            # do not change OPTIONS, omitted for brevity
        },
    },
]
```

Django lets us avoid hardcoding the path to `django/templates` by calculating the path to `django` at runtime as `BASE_DIR`. This way, we can use the same setting across environments.

Another important setting we just saw was `APP_DIRS`. This setting tells Django to check each installed app for a `templates` directory when Django is looking for a template. It means that we don't have to update the `DIRS` key for each installed app and lets us isolate our templates under our apps (increasing reusability). Finally, it's important to remember that apps are searched in the order they appear in `INSTALLED_APPS`. If there's a template name collision (for example, two apps provide a template called `registration/login.html`), then the one listed first in `INSTALLED_APPS` will be used.

Next, let's configure our project to use Bootstrap 4 when rendering forms in HTML.

Configuring Django Crispy Forms to use Bootstrap 4

In our base template, we included the Bootstrap 4 css template. To make it easy to render a form and style it using Bootstrap 4, we will use a third-party Django app called Django Crispy Forms. However, we must configure Django Crispy Forms to tell it to use Bootstrap 4.

Let's add a new setting to the bottom of `django/config/settings.py`:

```
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

Now, Django Crispy Forms is configured to use Bootstrap 4 when rendering a form. We'll take a look at it later in this chapter, in sections covering rendering a form in a template.

Next, let's create a mixin that ensures that only the owners of a mailing list can affect them.

Creating a mixin to check whether a user can use the mailing list

Django uses **class-based views** (**CBVs**) to make it easier to reuse code, simplifying repetitive tasks. One of the repetitive tasks we'll have to do in the `mailinglist` app is protect `MailingList` s and their related models from being tampered with by other users. We'll create a mixin that provides protection.

A mixin is a class that provides a limited functionality that is meant to be used in conjunction with other classes. We've previously seen the `LoginRequired` mixin, which can be used in conjunction with a view class to protect a view from unauthenticated access. In this section, we will create a new mixin.

Let's create our `UserCanUseMailingList` mixin in a new file at `django/mailinglist/mixins.py`:

```
from django.core.exceptions import PermissionDenied, FieldDoesNotExist

from mailinglist.models import MailingList


class UserCanUseMailingList:

    def get_object(self, queryset=None):
```

```
        obj = super().get_object(queryset)
        user = self.request.user
        if isinstance(obj, MailingList):
            if obj.user_can_use_mailing_list(user):
                return obj
            else:
                raise PermissionDenied()

        mailing_list_attr = getattr(obj, 'mailing_list')
        if isinstance(mailing_list_attr, MailingList):
            if mailing_list_attr.user_can_use_mailing_list(user):
                return obj
            else:
                raise PermissionDenied()
        raise FieldDoesNotExist('view does not know how to get mailing '
                                'list.')
```

Our class defines a single method, `get_object(self, queryset=None)`. This method has the same signature as `SingleObjectMixin.get_object()`, which is used by many of Django's built-in CBVs (for example, `DetailView`). Our `get_object()` implementation doesn't do any work to retrieve an object. Instead, our `get_object` just checks the object that a parent retrieved to check whether it is, or has, a `MailingList` and confirms that the logged in user can use the mailing list.

One surprising thing about a mixin is that it relies on a super class but doesn't inherit from one. In `get_object()`, we explicitly call `super()`, but `UserCanUseMailingList` doesn't have any base classes. Mixin classes aren't expected to be used by themselves. Instead, they will be used by classes, which subclass them *and* one or more other classes.

We'll take a look at how this works in the next few sections.

Creating MailingList views and templates

Now, we'll take a look at the views that will process the user's requests and return responses that show a UI created from our templates.

Let's start by creating a view to list of all our `MailingList`s.

Creating the MailingListListView view

We will create a view that shows the mailing lists a user owns.

Let's create our `MailingListListView` in `django/mailinglist/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView

from mailinglist.models import  MailingList


class MailingListListView(LoginRequiredMixin, ListView):

    def get_queryset(self):
        return MailingList.objects.filter(owner=self.request.user)
```

Our view is derived from two views, `LoginRequiredMixin` and `ListView`. `LoginRequiredMixin` is a mixin that ensures that a request made by an unauthenticated user is redirected to a login view instead of being processed. To help the `ListView` know *what* to list, we will override the `get_queryset()` method and return a `QuerySet` that includes only the `MailingList` s owned by the currently logged in user. To display the result, `ListView` will try to render a template at `appname/modelname_list.html`. In our case, `ListView` will try to render `mailinglist/mailinglist_list.html`.

Let's create that template in `django/mailinglist/templates/mailinglist/mailinglist_list.html`:

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

{% block title %}
  Your Mailing Lists
{% endblock %}

{% block body %}
  <div class="row user-mailing-lists" >
    <div class="col-sm-12" >
      <h1 >Your Mailing Lists</h1 >
      <div >
        <a class="btn btn-primary"
           href="{% url "mailinglist:create_mailinglist" %}" >New List</a >
      </div >
      <p > Your mailing lists:</p >
      <ul class="mailing-list-list">
        {% for mailinglist in mailinglist_list %}
          <li class="mailinglist-item">
```

```
            <a href="{% url "mailinglist:manage_mailinglist"
pk=mailinglist.id %}" >
              {{ mailinglist.name }}
            </a >
          </li >
        {% endfor %}
      </ul >
    </div >
  </div >
{% endblock %}
```

Our template extends `base.html`. When a template extends another template, it can only put HTML into the `block`s that have been previously defined. We will also see a lot of new Django template tags. Let's take a closer look at them:

- `{% extends "base.html" %}`: This tells the Django template language which template that we're extending.
- `{% block title %}... {% endblock %}`: This tells Django that we're providing new code that it should place in the extended template's `title` block. The previous code in that block (if any) is replaced.
- `{% for mailinglist in mailinglist_list %} ... {% endfor %}`: This provides a for loop for each item in the list.
- `{% url ... %}`: The `url` tag will produce a URL path for the named `path`.
- `{% url ... pk=...%}`: This works just like the preceding point, but, in some cases, a `path` may take arguments (for example, the primary key of the `MailingList` to display). We can specify these extra arguments in the `url` tag after the name of the `path`.

We now have a view and template that work together.

The final step with any view is adding the app's URLConf to it. Let's update `django/mailinglist/urls.py`:

```
from django.urls import path

from mailinglist import views

app_name = 'mailinglist'

urlpatterns = [
    path('',
         views.MailingListListView.as_view(),
         name='mailinglist_list'),
]
```

Given how we configured our root URLConf earlier, any request sent to `/mailinglist/` will be routed to our `MailingListListView`.

Next, let's add a view to create new `MailingList`s.

Creating the CreateMailingListView and template

We will create a view to create mailing lists. When our view receives a `GET` request, the view will show our users a form for entering the name of the mailing list. When our view receives a `POST` request, the view will validate the form and either redisplay the form with errors or create the mailing list and redirect the user to the list's management page.

Let's create the view now in `django/mailinglist/views.py`:

```
class CreateMailingListView(LoginRequiredMixin, CreateView):
    form_class = MailingListForm
    template_name = 'mailinglist/mailinglist_form.html'

    def get_initial(self):
        return {
            'owner': self.request.user.id,
        }
```

`CreateMailingListView` is derived from two classes:

- `LoginRequiredMixin` redirects requests that are not associated with a logged in user from being processed (we'll configure this later in this chapter, in the *Creating the user app* section)
- `CreateView` knows how to work with the form indicated in `form_class` and render it using the template listed in `template_name`

`CreateView` is the class that does most of the work here without us needing to provide almost any extra information. Processing a form, validating it, and saving it are always the same, and `CreateView` has the code to do it. If we need to change some of the behavior, we can override one for the hooks that `CreateView` provides, as we do with `get_initial()`.

When `CreateView` instantiates our `MailingListForm`, `CreateView` calls its `get_initial()` method to get the `initial` data (if any) for the form. We use this hook to make sure that the form's owner is set to the logged in user's `id`. Remember that `MailingListForm` has its `owner` field disabled, so the form will ignore any data provided by the user.

Next, let's create the template for our `CreateView` in `django/mailinglist/templates/mailinglist/mailinglist_form.html`:

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

{% load crispy_forms_tags %}

{% block title %}
  Create Mailing List
{% endblock %}

{% block body %}
  <h1 >Create Mailing List</h1 >
  <form method="post" class="col-sm-4" >
    {% csrf_token %}
    {{ form | crispy }}
    <button class="btn btn-primary" type="submit" >Submit</button >
  </form >
{% endblock %}
```

Our template extends `base.html`. When a template extends another template, it can only put HTML into the blocks that have been previously defined by the extended template(s). We also take a lot of new Django template tags. Let's take a closer look at them:

- `{% load crispy_forms_tags %}`: This tells Django to load a new template tag library. In this case, we will load `crispy_from_tags` from the Django Crispy Forms app that we have installed. This provides us with the `crispy` filter we'll see later in this section.
- `{% csrf_token %}`: Any form that Django processes must have a valid CSRF token to prevent CSRF attacks (refer to `Chapter 3`, *Posters, Headshots, and Security*). The `csrf_token` tag returns a hidden input tag with the correct CSRF token. Remember that Django generally won't process a POST request without a CSRF Token.
- `{{ form | crispy }}`: The `form` variable is a reference to the form instance that our view is processing and is passed into this template's context by our `CreateView`. `crispy` is a filter provided by the `crispy_form_tags` tag library and will output the form using HTML tags and CSS classes used in Bootstrap 4.

We now have a view and template that work together. The view is able to use the template to create a user interface to enter data into the form. The view is then able to process the form's data and create a `MailingList` model from valid form data or redisplay the form if the data has a problem. The Django Crispy Forms library renders the form using the HTML and CSS from the Bootstrap 4 CSS Framework.

Finally, let's add our view to the `mailinglist` app's URLConf. In `django/mailinglist/urls.py`, let's add a new `path()` object to the URLConf:

```
    path('new',
         views.CreateMailingListView.as_view(),
         name='create_mailinglist')
```

Given how we configured our root URLConf earlier, any request sent to `/mailinglist/new` will be routed to our `CreatingMailingListView`.

Next, let's make a view to delete a `MailingList`.

Creating the DeleteMailingListView view

Users will want to delete `MailingList` s after they stop being useful. Let's create a view that will prompt the user for confirmation on a `GET` request and delete the `MailingList` on a `POST`.

We'll add our view to `django/mailinglist/views.py`:

```
class DeleteMailingListView(LoginRequiredMixin, UserCanUseMailingList,
                            DeleteView):
    model = MailingList
    success_url = reverse_lazy('mailinglist:mailinglist_list')
```

Let's take a closer look at the classes that `DeleteMailingListView` is derived from:

- `LoginRequiredMixin`: This serves the same function as in the preceding code, ensuring that requests from an unauthenticated user aren't processed. The user is just redirected to the login page.
- `UserCanUseMailingList`: This is the mixin we created in the preceding code. `DeleteView` uses the `get_object()` method to retrieve the model instance to be deleted. By mixing `UserCanUseMailingList` into the `DeleteMailingListView` class, we protect each user's `MailingLists` from being deleted by unauthorized users.

- `DeleteView`: This is a Django view that knows how to render a confirmation template on a `GET` request and delete the related model on `POST`.

In order for Django's `DeleteView` to function properly, we will need to configure it properly. `DeleteView` knows which model to delete from its `model` attribute. `DeleteView` requires that we provide a `pk` argument when we route requests to it. To render the confirmation template, `DeleteView` will try to use `appname/modelname_confirm_delete.html`. In the case of `DeleteMailingListView`, the template will be `mailinglist/mailinglist_confirm_delete.html`. If the model is successfully deleted, then `DeleteView` will redirect to the `success_url` value. We've avoided hardcoding the `success_url` and instead used `reverse_lazy()` to refer to the URL by name. The `reverse_lazy()` function returns a value that won't resolve until it's used to create a `Response` object.

Let's create the template that `DeleteMailingListView` requires in `django/mailinglist/templates/mailinglist/mailinglist_confirm_delete.html`:

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

{% block title %}
  Confirm delete {{ mailinglist.name }}
{% endblock %}

{% block body %}
  <h1 >Confirm Delete?</h1 >
  <form action="" method="post" >
    {% csrf_token %}
    <p >Are you sure you want to delete {{ mailinglist.name }}?</p >
    <input type="submit" value="Yes" class="btn btn-danger btn-sm ">
    <a class="btn btn-primary btn-lg" href="{% url
"mailinglist:manage_mailinglist" pk=mailinglist.id %}">No</a>
  </form >
{% endblock %}
```

In this template, we don't use any forms because there isn't any input to validate. The form submission itself is the confirmation.

The last step will be adding our view to the `urlpatterns` list in `django/mailinglist/urls.py`:

```
    path('<uuid:pk>/delete',
        views.DeleteMailingListView.as_view(),
        name='delete_mailinglist'),
```

This `path` looks different than the previous `path()` calls we've seen. In this `path`, we're including a named argument that will be parsed out of the path and passed to the view. We specify `path` named arguments using the `<converter:name>` format. A converter knows how to match a part of the path (for example, the `uuid` converter knows how to match a UUID; `int` knows how to match a number; `str` will match any non-empty string except `/`). The matched text is then passed to the view as a key word argument with the provided name. In our case, to route a request to `DeleteMailingListView`, it has to have a path like this: `/mailinglist/bce93fec-f9c6-4ea7-b1aa-348d3bed4257/delete`.

Now that we can list, create, and delete `MailingList`s, let's create a view to manage its `Subscriber`s and `Message`s.

Creating MailingListDetailView

Let's create a view that will list all the `Subscriber`s and `Message`s related to a `MailingList`. We want also need a place to show our users the `MailingList`s subscription page link. Django can make it easy to create a view that represents a model instance.

Let's create our `MailingListDetailView` in `django/mailinglist/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import DetailView

from mailinglist.mixins import UserCanUseMailingList
from mailinglist.models import MailingList


class MailingListDetailView(LoginRequiredMixin, UserCanUseMailingList,
                            DetailView):
    model = MailingList
```

We're using the `LoginRequiredMixin` and `UserCanUseMailingList` the same way and for the same purpose as before. This time, we're using them with `DetailView`, which is one of the simplest views. It simply renders a template for an instance of the model it's been configured for. It retrieves the model instance by receiving a `pk` argument from `path` just like `DeleteView`. Also, we don't have to explicitly configure the template it will use because, by convention, it uses `appname/modelname_detail.html`. In our case, it will be `mailinglist/mailinglist_detail.html`.

Let's create our template in `django/mailinglist/templates/mailinglist/mailinglist_detail.html`:

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

{% block title %}
  {{ mailinglist.name }} Management
{% endblock %}

{% block body %}
  <h1 >{{ mailinglist.name }} Management
    <a class="btn btn-danger"
       href="{% url "mailinglist:delete_mailinglist" pk=mailinglist.id %}"
>
      Delete</a >
  </h1 >

  <div >
    <a href="{% url "mailinglist:create_subscriber"
mailinglist_pk=mailinglist.id %}" >Subscription
      Link</a >

  </div >

  <h2 >Messages</h2 >
  <div > Send new
    <a class="btn btn-primary"
       href="{% url "mailinglist:create_message"
mailinglist_pk=mailinglist.id %}">
      Send new Message</a >
  </div >
  <ul >
    {% for message in mailinglist.message_set.all %}
      <li >
        <a href="{% url "mailinglist:view_message" pk=message.id %}" >{{
message.subject }}</a >
      </li >
    {% endfor %}
```

```
    </ul >

    <h2 >Subscribers</h2 >
    <ul >
      {% for subscriber in mailinglist.subscriber_set.all %}
        <li >
          {{ subscriber.email }}
          {{ subscriber.confirmed|yesno:"confirmed,unconfirmed" }}
          <a href="{% url "mailinglist:unsubscribe" pk=subscriber.id %}" >
            Unsubscribe
          </a >
        </li >
      {% endfor %}
    </ul >
  {% endblock %}
```

The preceding code template introduces only one new item (the `yesno` filter), but really shows how all the tools of Django's template language come together.

The `yesno` filter takes a value and returns `yes` if the value evaluates to `True`, `no` if it evaluates to `False`, and `maybe` if it is `None`. In our case, we've passed an argument that tells `yesno` to return `confirmed` if `True` and `unconfirmed` if `False`.

The `MailingListDetailView` class and template illustrate how Django lets us concisely complete a common web developer task: display a page for a row in a database.

Next, let's create a new `path()` object to our view in the `mailinglist` URLConf:

```
    path('<uuid:pk>/manage',
         views.MailingListDetailView.as_view(),
         name='manage_mailinglist')
```

Next, let's create views for our `Subscriber` model instances.

Creating Subscriber views and templates

In this section we'll create views and templates to let users interact with our `Subscriber` model. One of the main differences between these views and the `MailingList` and `Message` views is that they will not need any mixins because they will be exposed publicly. Their main protection from tampering is that `Subscriber`s are identified by a UUID which has a large key space, meaning that tampering is unlikely.

Let's start with `SubscribeToMailingListView`.

Creating SubscribeToMailingListView and template

We need a view to collect `Subscriber`s to `MailingList`s. Let's create a `SubscribeToMailingListView` class with `django/mailinglist/views.py`:

```
class SubscribeToMailingListView(CreateView):
    form_class = SubscriberForm
    template_name = 'mailinglist/subscriber_form.html'

    def get_initial(self):
        return {
            'mailing_list': self.kwargs['mailinglist_id']
        }

    def get_success_url(self):
        return reverse('mailinglist:subscriber_thankyou', kwargs={
            'pk': self.object.mailing_list.id,
        })

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        mailing_list_id = self.kwargs['mailinglist_id']
        ctx['mailing_list'] = get_object_or_404(
            MailingList,
            id=mailing_list_id)
        return ctx
```

Our `SubscribeToMailingListView` is similar to `CreateMailingListView` but overrides a couple of new methods:

- `get_success_url()`: This is called by `CreateView` to get a URL to redirect the user to the model that has been created. In `CreateMailingListView`, we didn't need to override it because the default behavior uses the model's `get_absolute_url`. We use the `reverse()` function resolve the path to the thank you page.
- `get_context_data()`: This lets us add new variables to the template's context. In this case, we need access to the `MailingList` the user may subscribe to show the `MailingList`'s name. We use Django's `get_object_or_404()` shortcut function to retrieve the `MailingList` by its ID or raise a 404 exception. We'll have this view's `path` parse the `mailinglist_id` out of our request's path (refer to to the , at the end of this section).

Next, let's create our template in `mailinglist/templates/mailinglist/subscriber_form.html`:

```
{% extends "base.html" %}
{% load crispy_forms_tags %}
{% block title %}
Subscribe to {{ mailing_list }}
{% endblock %}

{% block body %}
<h1>Subscribe to {{ mailing_list }}</h1>
<form method="post" class="col-sm-6 ">
  {% csrf_token %}
  {{ form | crispy }}
  <button class="btn btn-primary" type="submit">Submit</button>
</form>
{% endblock %}
```

This template doesn't introduce any tags but shows another example of how we can use Django's template language and the Django Crispy Forms API to quickly build a pretty HTML form. We extend `base.html`, as before, to give our page a consistent look and feel. `base.html` also provides the blocks we're going to put our content into. Outside of any block, we `{% load %}` the Django Crispy Forms tag library so that we can use the `crispy` filter on our form to generate the Bootstrap 4 compatible HTML.

Next, let's make sure that Django knows how to route requests to our new view by adding a `path()` to `SubscribeToMailingListView` to the `mailinglist` app's URLConf's `urlpatterns` list:

```
    path('<uuid:mailinglist_id>/subscribe',
         views.SubscribeToMailingListView.as_view(),
         name='subscribe'),
```

In this `path()`, we need to match the `uuid` parameter that we pass to our view as `mailinglist_pk`. This is the keyword argument that our `get_context_data()` method referenced.

Next, let's create a thank you page to thank users for subscribing to a mailing list.

Creating a thank you for subscribing view

After a user subscribes to a mailing list, we want to show them a *thank you* page. This page can be the same for all users who subscribe to the same mailing list since all it will show is the name of the mailing list (not the subscriber's email). To create this view, we're going to use the `DetailView` we've seen before but this time without any additional mixing (there's no information to protect here).

Let's create our `ThankYouForSubscribingView` in `django/mailinglist/views.py`:

```
from django.views.generic import DetailView

from mailinglist.models import  MailingList


class ThankYouForSubscribingView(DetailView):
    model = MailingList
    template_name = 'mailinglist/subscription_thankyou.html'
```

Django does all the work for us in the `DetailView` as long as we provide a `model` attribute. The `DetailView` knows how to look up a model and then render a template for that model. We also provide a `template_name` attribute because the `mailinglist/mailinglist_detail.html` template (which `DetailView` would use by default) is already being used by `MailingListDetailView`.

Let's create our template in `django/mailinglist/templates/mailinglist/subscription_thankyou.html`:

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

{% block title %}
  Thank you for subscribing to {{ mailinglist }}
{% endblock %}

{% block body %}
  <div class="col-sm-12" ><h1 >Thank you for subscribing
    to {{ mailinglist }}</h1 >
    <p >Check your email for a confirmation email.</p >
  </div >
{% endblock %}
```

Our template just shows a thank you and the template name.

Finally, let's add a `path()` to `ThankYouForSubscribingView` to the `mailinglist` app's URLConf's `urlpatterns` list:

```
path('<uuid:pk>/thankyou',
     views.ThankYouForSubscribingView.as_view(),
     name='subscriber_thankyou'),
```

Our `path` needs to match a UUID in order to route a request to `ThankYouForSubscribingView`. The UUID will be passed into the view as the keyword argument `pk`. This `pk` will be used by `DetailView` to find the correcting `MailingList`.

Next, we will need to let a user confirm that they want to receive emails at this address.

Creating a subscription confirmation view

To prevent spammers from abusing our service, we will need to send an email to our subscribers to confirm that they really want to subscribe to one of our users' mailing lists. We'll cover sending those emails, but we'll create the confirmation page now.

This confirmation page will behave a little strangely. Simply visiting the page will modify `Subscriber.confirmed` to `True`. This is standard for how mailing list confirmation pages work (we want to avoid creating extra work for our subscribers) but strange according to the HTTP spec, which says that `GET` requests should not modify a resource.

Let's create our `ConfirmSubscriptionView` in `django/mailinglist/views.py`:

```
from django.views.generic import DetailView

from mailinglist.models import  Subscriber


class ConfirmSubscriptionView(DetailView):
    model = Subscriber
    template_name = 'mailinglist/confirm_subscription.html'

    def get_object(self, queryset=None):
        subscriber = super().get_object(queryset=queryset)
        subscriber.confirmed = True
        subscriber.save()
        return subscriber
```

`ConfirmSubscriptionView` is another `DetailView` since it shows a single model instance. In this case, we override the `get_object()` method in order to modify the object before returning it. Since `Subscriber`s are not required to be users of our system, we don't need to use `LoginRequiredMixin`. Our view is protected from brute force enumeration because the key space of `Subscriber.id` is large and assigned non-sequentially.

Next, let's create our template in `django/mailinglist/templates/mailinglist/confirm_subscription.html`:

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

{% block title %}
  Subscription to {{ subscriber.mailing_list }} confirmed.
{% endblock %}

{% block body %}
  <h1 >Subscription to {{ subscriber.mailing_list }} confirmed!</h1 >
{% endblock %}
```

Our template uses the blocks defined in `base.html` to simply notify the user of their confirmed subscription.

Finally, let's add a `path()` to `ConfirmSubscriptionView` to the `mailinglist` app's URLConf's `urlpatterns` list:

```
    path('subscribe/confirmation/<uuid:pk>',
         views.ConfirmSubscriptionView.as_view(),
         name='confirm_subscription')
```

Our `confirm_subscription` path defines the path to match in order to route a request to our view. Our matching expression includes the requirement of a UUID, which will be passed to our `ConfirmSubscriptionView` as the keyword argument `pk`. The parent (`DetailView`) of `ConfirmSubscriptionView` will then use that to retrieve the correct `Subscriber`.

Next, let's allow `Subscribers` to unsubscribe themselves.

Creating UnsubscribeView

Part of being an ethical mailing provider is letting our `Subscriber`s unsubscribe. Next, we'll create an `UnsubscribeView`, which will delete a `Subscriber` model instance after they've confirmed they definitely want to unsubscribe.

Let's add our view to `django/mailinglist/views.py`:

```
from django.views.generic import DeleteView

from mailinglist.models import Subscriber


class UnsubscribeView(DeleteView):
    model = Subscriber
    template_name = 'mailinglist/unsubscribe.html'

    def get_success_url(self):
        mailing_list = self.object.mailing_list
        return reverse('mailinglist:subscribe', kwargs={
            'mailinglist_pk': mailing_list.id
        })
```

Our `UnsubscribeView` lets Django's built-in `DeleteView` implement to render the template and find and delete the correct `Subscriber`. `DeleteView` requires that it receive a `pk` for the `Subscriber` as a keyword argument parsed from the path (much like a `DetailView`). When the delete succeeds, we'll redirect the user to the subscription page with the `get_success_url()` method. When `get_success_url()` is executing, our `Subscriber` instance will already be deleted from the database, but a copy of the respective object will be available under `self.object`. We will use that still in memory (but not in the database) instance to get the `id` attribute of the related mailing list.

To render the confirmation form, we will need to create a template in `django/mailinglist/templates/mailinglist/unsubscribe.html`:

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

{% block title %}
  Unsubscribe?
{% endblock %}

{% block body %}
  <div class="col">
    <form action="" method="post" >
      {% csrf_token %}
      <p >Are you sure you want to unsubscribe
        from {{ subscriber.mailing_list.name }}?</p >
      <input class="btn btn-danger" type="submit"
             value="Yes, I want to unsubscribe " >
    </form >
  </div >
{% endblock %}
```

This template renders a `POST` form, which will act as confirmation of the desire of the `subscriber` to be unsubscribed.

Next, let's add a `path()` to `UnsubscribeView` to the `mailinglist` app's URLConf's `urlpatterns` list:

```
        path('unsubscribe/<uuid:pk>',
             views.UnsubscribeView.as_view(),
             name='unsubscribe'),
```

When dealing with views that derive from `DetailView` or `DeleteView`, it's vital to remember to name the path matcher `pk`.

Great, now, let's allow the user to start creating `Message`s that they will send to their `Subscriber`s.

Creating Message Views

We track the emails that our users want to send to their `Subscriber`s in the `Message` model. To make sure we have an accurate log of what users send to their `Subscribers`, we will restrict the operations available on `Message`s. Our users will only be able to create and view `Message`s. It doesn't make sense to support editing since an email that's been sent can't be modified. We also won't support deleting messages so that both we and the users have an accurate log of what was requested to be sent when.

Let's start with making a `CreateMessageView`!

Creating CreateMessageView

Our `CreateMessageView` is going to follow a pattern similar to the markdown forms that we created for Answerly. The user will get a form that they can submit to either save or preview. If the submit is a preview, then the form will render along with the preview of the rendered markdown of the `Message`. If the user chooses save, then they will create their new message.

Since we're creating a new model instance, we will use Django's `CreateView`.

Let's create our view in `django/mailinglist/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView

from mailinglist.models import Message


class CreateMessageView(LoginRequiredMixin, CreateView):
    SAVE_ACTION = 'save'
    PREVIEW_ACTION = 'preview'

    form_class = MessageForm
    template_name = 'mailinglist/message_form.html'

    def get_success_url(self):
        return reverse('mailinglist:manage_mailinglist',
                       kwargs={'pk': self.object.mailing_list.id})

    def get_initial(self):
        mailing_list = self.get_mailing_list()
        return {
            'mailing_list': mailing_list.id,
        }

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        mailing_list = self.get_mailing_list()
        ctx.update({
            'mailing_list': mailing_list,
            'SAVE_ACTION': self.SAVE_ACTION,
            'PREVIEW_ACTION': self.PREVIEW_ACTION,
        })
        return ctx

    def form_valid(self, form):
        action = self.request.POST.get('action')
        if action == self.PREVIEW_ACTION:
            context = self.get_context_data(
                form=form,
                message=form.instance)
            return self.render_to_response(context=context)
        elif action == self.SAVE_ACTION:
            return super().form_valid(form)
```

```
def get_mailing_list(self):
    mailing_list = get_object_or_404(MailingList,
                                     id=self.kwargs['mailinglist_pk'])
    if not mailing_list.user_can_use_mailing_list(self.request.user):
        raise PermissionDenied()
    return mailing_list
```

Our view inherits from `CreateView` and `LoginRequiredMixin`. We use the `LoginRequiredMixin` to prevent unauthenticated users from sending messages to mailing lists. To prevent logged in but unauthorized users from sending messages, we will create a central `get_mailing_list()` method, which checks that the logged in user can use this mailing list. `get_mailing_list()` expects that the `mailinglist_pk` will be provided as a keyword argument to the view.

Let's take a closer look at the `CreateMessageView` to see how this all works together:

- `form_class = MessageForm`: This is the form that we want `CreateView` to render, validate, and use to create our `Message` model.
- `template_name = 'mailinglist/message_form.html'`: This is the template that we'll create next.
- `def get_success_url()`: After a `Message` is successfully created, we'll redirect our users to the management page of the `MailingList`.
- `def get_initial()`:: Our `MessageForm` has its `mailing_list` field disabled so that users can't try to surreptitiously create a `Message` for another user's `MailingList`. Instead, we use our `get_mailing_list()` method to get the mailing list based on the `mailinglist_pk` argument. Using `get_mailing_list()`, we check whether the logged in user can use the `MailingList`.
- `def get_context_data()`: This provides extra variables to the template's context. We provide the `MailingList` as well as the save and preview constants.
- `def form_valid()`: This defines the behavior if the form is valid. We override the default behavior of `CreateView` to check the `action` POST argument. `action` will tell us whether to render a preview of the `Message` or to let `CreateView` save a new `Message` model instance. If we're previewing the message, then we pass an unsaved `Message` instance built by our form to the template's context.

Next, let's make our template in `django/mailinglist/templates/mailinglist/message_form.html`:

```
{% extends "base.html" %}
{% load crispy_forms_tags %}
{% load markdownify %}
{% block title %}
  Send a message to {{ mailing_list }}
{% endblock %}

{% block body %}
  <h1 >Send a message to {{ mailing_list.name }}</h1 >
  {% if message %}
    <div class="card" >
      <div class="card-header" >
        Message Preview
      </div >
      <div class="card-body" >
        <h5 class="card-title" >{{ message.subject }}</h5 >
        <div>{{ message.body|markdownify }}</div>
      </div >
    </div >
  {% endif %}
  <form method="post" class="col-sm-12 col-md-9" >
    {% csrf_token %}
    {{ form | crispy }}
    <button type="submit" name="action"
            value="{{ SAVE_ACTION }}"
            class="btn btn-primary" >Save
    </button >
    <button type="submit" name="action"
            value="{{ PREVIEW_ACTION }}"
            class="btn btn-primary" >Preview
    </button >
  </form >
{% endblock %}
```

This template loads the third party Django Markdownify tag library and the Django Crispy Forms tag library. The former gives us the `markdownify` filter and the latter gives us the `crispy` filter. The `markdownify` filter will convert the markdown text it receives into HTML. We previously used Django Markdownify in our Answerly project in part 2.

This template form has two submit buttons, one to save the form and one to preview the form. The preview block is only rendered if we pass in `message` to preview.

Now that we have our view and template, let's add a `path()` to `CreateMessageView` in the `mailinglist` app's URLConf:

```
        path('<uuid:mailinglist_ipk>/message/new',
            views.CreateMessageView.as_view(),
            name='create_message'),
```

Now that we can create messages, let's make a view to view messages we've already created.

Creating the Message DetailView

To let users view the `Message`s they have sent to their `Subscriber`s we need a `MessageDetailView`. This view will simply display a `Message` but should only let users who are logged in and can use the `Message`'s `MailingList` access the view.

Let's create our view in `django/mailinglist/views.py`:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import DetailView

from mailinglist.mixins import UserCanUseMailingList
from mailinglist.models import Message


class MessageDetailView(LoginRequiredMixin, UserCanUseMailingList,
                        DetailView):
    model = Message
```

As the name implies, we're going to use the Django's `DetailView`. To provide the protection we need, we'll add Django's `LoginRequiredMixin` and our `UserCanUseMailingList` mixin. As we've seen before, we don't need to specify the name of the template because `DetailView` will assume it based on the name of the app and model. In our case, `DetailView` wants the template to be called `mailinglist/message_detail.html`.

Let's create our template in `mailinglist/message_detail.html`:

```
{% extends "base.html" %}
{% load markdownify %}

{% block title %}
  {{ message.subject }}
{% endblock %}
```

```
{% block body %}
  <h1 >{{ message.subject }}</h1 >
  <div>
    {{ message.body|markdownify }}
  </div>
{% endblock %}
```

Our template extends `base.html` and shows the message in the `body` block. When showing the `Message.body`, we use the third party Django Markdownify tag library's `markdownify` filter to render any markdown text as HTML.

Finally, we need to add a `path()` to `MessageDetailView` to the `mailinglist` app's URLConf's `urlpatterns` list:

```
path('message/<uuid:pk>',
     views.MessageDetailView.as_view(),
     name='view_message')
```

We've now completed our `mailinglist` app's models, views, and templates. We've even created a `UserCanUseMailingList` to let our views easily prevent unauthorized access to a `MailingList` or one of its related views.

Next, we'll create a `user` app to encapsulate user registration and authentication.

Creating the user app

To create a `MailingList` in Mail Ape, the user needs to have an account and be logged in. In this section, we will write the code for our `user` Django app, which will encapsulate everything to do with a user. Remember that the Django app should be tightly scoped. We don't want to put this behavior in our `mailinglist` app, as these are two discrete concerns.

Our `user` app is going to be very similar to the `user` app seen in MyMDB (Part 1) and Answerly (Part 2). Due to this similarity, we will gloss over some topics. For a deeper examination of the topic, refer to `Chapter 2`, *Adding Users to MyMDb*.

Django makes managing users and authentication easier with its built-in `auth` app (`django.contrib.auth`). The `auth` app offers a default user model, a `Form` for creating new users, as well as log in and log out views. This means that our `user` app only needs to fill in a few blanks before we have complete user management working locally.

Let's start by creating a URLConf for our `user` app in `django/user/urls.py`:

```
from django.contrib.auth.views import LoginView, LogoutView
from django.urls import path

import user.views

app_name = 'user'

urlpatterns = [
    path('login', LoginView.as_view(), name='login'),
    path('logout', LogoutView.as_view(), name='logout'),
    path('register', user.views.RegisterView.as_view(), name='register'),
]
```

Our URLConf is made up of three views:

- `LoginView.as_view()`: This is the `auth` app's login view. The `auth` app provides a view for accepting credentials but doesn't have a template. We'll need to create a template with the name `registration/login.html`. By default, it will redirect a user to `settings.LOGIN_REDIRECT_URL` on login. We can also pass a `next GET` parameter to supersede the setting.
- `LogoutView.as_view()`: This is the auth app's logout view. `LogoutView` is one of the few views that modifies state on a `GET` request, logging the user out. The view returns a redirect response. We can use `settings.LOGOUT_REDIRECT_URL` to configure where our user will be redirected to during log out. Again, we use the `GET` parameter `next` to customize this behavior.
- `user.views.RegisterView.as_view()`: This is the user registration view we will write. Django provides us with a `UserCreationForm` but not a view.

We also need to add a few settings to make Django use our `user` view properly. Let's update `django/config/settings.py` with some new settings:

```
LOGIN_URL = 'user:login'
LOGIN_REDIRECT_URL = 'mailinglist:mailinglist_list'
LOGOUT_REDIRECT_URL = 'user:login'
```

These three settings tell Django how to redirect the user in different authentication scenarios:

- `LOGIN_URL`: When an unauthenticated user tries to access a page that requires authentication, `LoginRequiredMixin` uses this setting.
- `LOGIN_REDIRECT_URL`: When a user logs in, where should we redirect them to? Often, we redirect them to a profile page; in our case, the page that shows a list of `MailingLists`.
- `LOGOUT_REDIRECT_URL`: When a user logs out, where should we redirect them to? In our case, the login page.

We now have two more tasks left:

- Creating the login template
- Creating the user registration view and template

Let's start by making the login template.

Creating the login template

Let's make our login template in `django/user/templates/registration/login.html`:

```
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block title %} Login - {{ block.super }} {% endblock %}

{% block body %}
  <h1>Login</h1>
  <form method="post" class="col-sm-6">
    {% csrf_token %}
    {{ form|crispy }}
    <button type="submit" id="log_in" class="btn btn-primary">Log
in</button>
  </form>
{% endblock %}
```

This form follows all the practices of our previous forms. We use `csrf_token` to protect against a CSRF attack. We use the `crsipy` filter to print the form using Bootstrap 4 style tags and classes.

Remember, we didn't need to make a view to process our login requests because we're using the one that comes with `django.contrib.auth`.

Next, let's create a view and template to register new users.

Creating the user registration view

Django doesn't come with a view for creating new users, but it does offer a form for capturing a new user's registration. We can combine the `UserCreationForm` with a `CreateView` to quickly create a `RegisterView`.

Let's add our view to `django/user/views.py`:

```
from django.contrib.auth.forms import UserCreationForm
from django.views.generic.edit import CreateView


class RegisterView(CreateView):
    template_name = 'user/register.html'
    form_class = UserCreationForm
```

This is a very simple `CreateView`, like we've seen a few times in this chapter already.

Let's create our template in `django/user/templates/user/register.html`:

```
{% extends "base.html" %}
{% load crispy_forms_tags %}
{% block body %}
  <div class="col-sm-12">
    <h1 >Register for Mail Ape</h1 >
    <form method="post" >
      {% csrf_token %}
      {{ form | crispy }}
      <button type="submit" class="btn btn-primary" >
        Register
      </button >
    </form >
  </div >
{% endblock %}
```

Again, the template follows the same pattern as our previous `CreateView` templates.

Now, we're ready to run Mail Ape locally.

Running Mail Ape locally

Django comes with a development server. This server is not suitable for production (or even staging) deployment, but is suitable for local development.

Let's start the server using our Django project's `manage.py` script:

```
$ cd django
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
January 29, 2018 - 23:35:15
Django version 2.0.1, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

We can now access our server on `http://127.0.0.1:8000`.

Summary

In this chapter, we started our Mail Ape project. We created the Django project and started two Django apps. The `mailinglist` app contains our models, views, and templates for the mailing list code. The `user` app holds views and templates related to users. The `user` app is much simpler because it leverages Django's `django.contrib.auth` app.

Next, we'll build an API so that users can integrate with Mail Ape easily.

11
The Task of Sending Emails

Now that we have our models and views, we will need to make Mail Ape send emails. We'll have Mail Ape send two kinds of emails, subscriber confirmation emails and mailing list messages. We'll track mailing list message success by creating a new model called `SubscriberMessage` to track whether a message was successfully sent to an address stored in a `Subscriber` model instance. Since sending emails to a lot of `Subscriber` model instances can take a lot of time, we'll use Celery to send emails as tasks outside the regular Django request/response cycle.

In this chapter, we will do the following things:

- Use Django's template system to generate the HTML body of our emails
- Send emails that include both HTML and plain text using Django
- Use Celery to execute asynchronous tasks
- Prevent our code from sending actual emails during testing

Let's start by creating some common resources that we'll use to send dynamic emails.

Creating common resources for emails

In this section, we will create a base HTML email template and a `Context` object for rendering email templates. We want to create a base HTML template for our emails so that we can avoid repeating boilerplate HTML. We also want to make sure that every email we send includes an unsubscribe link to be good email citizens. Our `EmailTemplateContext` class will consistently provide the common variables that our templates need.

Let's start by creating a base HTML email template.

Creating the base HTML email template

We'll create our base email HTML template in `django/mailinglist/templates/mailinglist/email/base.html`:

```
<!DOCTYPE html>
<html lang="en" >
<head >
<body >
{% block body %}
{% endblock %}

Click <a href="{{ unsubscription_link }}">here</a> to unsubscribe from this
mailing list.
Sent with Mail Ape  .
</body >
</html >
```

The preceding template looks like a much simpler version of `base.html`, except it has only one block. Email templates can extend `email/base.html` and override the body block to avoid the boilerplate HTML. Despite the filenames being the same (`base.html`), Django won't confuse the two. Templates are identified by their template paths, not just filenames.

Our base template also expects the `unsubscription_link` variable to always exist. This will let users unsubscribe if they don't want to continue receiving emails.

To make sure that our templates always have the `unsubscription_link` variable, we'll create a `Context` that makes sure to always provide it.

Creating EmailTemplateContext

As we've discussed before (refer to `Chapter 1`, *Building MyMDB*), to render a template, we will need to provide Django with a `Context` object that has the variables the template references. When writing class-based views, we only have to provide a dict in the `get_context_data()` method and Django takes care of everything for us. However, when we want to render a template ourselves, we'll have to instantiate the `Context` class ourselves. To ensure that all our email template-rendering code provides the same minimum information, we'll create a custom template `Context`.

Let's create our `EmailTemplateContext` class in `django/mailinglist/emails.py`:

```
from django.conf import settings

from django.template import Context


class EmailTemplateContext(Context):

    @staticmethod
    def make_link(path):
        return settings.MAILING_LIST_LINK_DOMAIN + path

    def __init__(self, subscriber, dict_=None, **kwargs):
        if dict_ is None:
            dict_ = {}
        email_ctx = self.common_context(subscriber)
        email_ctx.update(dict_)
        super().__init__(email_ctx, **kwargs)

    def common_context(self, subscriber):
        subscriber_pk_kwargs = {'pk': subscriber.id}
        unsubscribe_path = reverse('mailinglist:unsubscribe',
                                   kwargs=subscriber_pk_kwargs)
        return {
            'subscriber': subscriber,
            'mailing_list': subscriber.mailing_list,
            'unsubscribe_link': self.make_link(unsubscribe_path),
        }
```

Our `EmailTemplateContext` is made up of the following three methods:

- `make_link()`: This joins a URL's path with our project's `MAILING_LIST_LINK_DOMAIN` setting. The `make_link` is necessary because Django's `reverse()` function doesn't include a domain. A Django project can be hosted on multiple different domains. We'll discuss the `MAILING_LIST_LINK_DOMAIN` value more in the *Configuring email settings* section.
- `__init__()`: This overrides the `Context.__init__(...)` method to give us a chance to add the results of the `common_context()` method to the value of the `dict_` parameter. We're careful to let the data received by the argument overwrite the data we generate in `common_context`.

- `common_context()`: This returns a dictionary that provides the variables we want available to all `EmailTemplateContext` objects. We always want to have `subscriber`, `mailing_list`, and `unsubscribtion_link` available.

We'll use both these resources in our next section, where we'll send confirmation emails to new `Subscriber` model instances.

Sending confirmation emails

In this section, we'll send emails to new `Subscriber`s to let them confirm their subscription to a `MailingList`.

In this section, we will:

1. Add Django's email configuration settings to our `settings.py`
2. Write a function to send emails using Django's `send_mail()` function
3. Create and render HTML and text templates for the body of our emails
4. Update `Subscriber.save()` to send the emails when a new `Subscriber` is created

Let's start by updating configuration with our mail server's settings.

Configuring email settings

In order to be able to send emails, we need to configure Django to talk to a **Simple Mail Transfer Protocol** (**SMTP**) server. In development and while learning, you can probably use the same SMTP server that your email client uses. Using such a server for sending large amounts of production email is likely a violation of your email provider's Terms of Service and can lead to account suspension. Be careful of which accounts you use.

Let's update our settings in `django/config/settings.py`:

```
EMAIL_HOST = 'smtp.example.com'
EMAIL_HOST_USER = 'username'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_PASSWORD = os.getenv('EMAIL_PASSWORD')

MAILING_LIST_FROM_EMAIL = 'noreply@example.com'
MAILING_LIST_LINK_DOMAIN = 'http://localhost:8000'
```

In the preceding code sample, I've used a lot of instances of `example.com`, which you should replace with the correct domain for your SMTP host and your domain. Let's take a closer look at the settings:

- `EMAIL_HOST`: This is the address of the SMTP sever we're using.
- `EMAIL_HOST_USER`: The username used to authenticate to the SMTP server.
- `EMAIL_PORT`: The port to connect to the SMTP server.
- `EMAIL_USE_TLS`: This is optional and defaults to `False`. Use it if you're connecting over TLS to the SMTP server. If you're using SSL, then use the `EMAIL_USE_SSL` setting. The SSL and TLS settings are mutually exclusive.
- `EMAIL_HOST_PASSWORD`: The password for the host. In our case, we will expect the password in an environment variable.
- `MAILING_LIST_FROM_EMAIL`: This is a custom setting we're using to set who set the `FROM` header on the emails we send.
- `MAILING_LIST_LINK_DOMAIN`: This is the domain to prefix all email template links with. We saw this setting used in our `EmailTemplateContext` class.

Next, let's write our create function to send the confirmation emails.

Creating the send email confirmation function

Now, we will create a function that will actually create and send confirmation emails to our `Subscriber`s. The `email` module will contain all our email-related code (we've already created the `EmailTemplateContext` class there).

Our `send_confirmation_email()` function will have to do the following:

1. Create a `Context` for rendering the email bodies
2. Create the subject for the email
3. Render the HTML and text email body
4. Send the email using the `send_mail()` function

Let's create that function in `django/mailinglist/emails.py`:

```
from django.conf import settings
from django.core.mail import send_mail
from django.template import engines, Context
from django.urls import reverse

CONFIRM_SUBSCRIPTION_HTML = 'mailinglist/email/confirmation.html'

CONFIRM_SUBSCRIPTION_TXT = 'mailinglist/email/confirmation.txt'


class EmailTemplateContext(Context):
    # skipped unchanged class


def send_confirmation_email(subscriber):
    mailing_list = subscriber.mailing_list
    confirmation_link = EmailTemplateContext.make_link(
        reverse('mailinglist:confirm_subscription',
                kwargs={'pk': subscriber.id}))
    context = EmailTemplateContext(
        subscriber,
        {'confirmation_link': confirmation_link}
    )
    subject = 'Confirming subscription to {}'.format(mailing_list.name)

    dt_engine = engines['django'].engine
    text_body_template = dt_engine.get_template(CONFIRM_SUBSCRIPTION_TXT)
    text_body = text_body_template.render(context=context)
    html_body_template = dt_engine.get_template(CONFIRM_SUBSCRIPTION_HTML)
    html_body = html_body_template.render(context=context)

    send_mail(
        subject=subject,
        message=text_body,
        from_email=settings.MAILING_LIST_FROM_EMAIL,
        recipient_list=(subscriber.email,),
        html_message=html_body)
```

Let's take a closer look at our code:

- `EmailTemplateContext()`: This instantiates the `Context` class we created earlier. We provide it with a `Subscriber` instance and a `dict`, which contains the confirmation link. The `confirmation_link` variable will be used by our templates, which we'll create in the next two sections.

- `engines['django'].engine`: This references the Django Template engine. The engine knows how to find `Templates` using the configuration settings in the `TEMPLATES` setting of `settings.py`.
- `dt_engine.get_template()`: This returns a template object. We provide the name of the template as an argument to the `get_template()` method.
- `text_body_template.render()`: This renders the template (using the context we created previously) into a string.

Finally, we send the email using the `send_email()` function.
The `send_email()` function takes the following arguments:

- `subject=subject`: The subject of the email message.
- `message=text_body`: The text version of the email.
- `from_email=settings.MAILING_LIST_FROM_EMAIL`: The sender's email address. If we don't provide a `from_email` argument, then Django will use the `DEFAULT_FROM_EMAIL` setting.
- `recipient_list=(subscriber.email,)`: A list (or tuple) of recipient email addresses. This must be a collection, even if you're only sending to one recipient. If you include multiple recipients, they will be able to see each other.
- `html_message=html_body`: The HTML version of the email. This argument is optional, as we don't have to provide an HTML body. If we provide an HTML body, then Django will send an email that includes both the HTML and text body. Email clients will choose to display the HTML or the plain text version of the email.

Now that we have our code for sending the emails, let's make our email body templates.

Creating the HTML confirmation email template

Let's make the HTML subscription email confirmation template. We'll create the template in `django/mailinglist/templates/mailinglist/email_templates/confirmation.html`:

```
{% extends "mailinglist/email_templates/email_base.html" %}

{% block body %}
  <h1>Confirming subscription to {{ mailing_list }}</h1 >
  <p>Someone (hopefully you) just subscribed to {{ mailinglist }}.</p >
  <p>To confirm your subscription click <a href="{{ confirmation_link
}}">here</a>.</p >
```

```
  <p>If you don't confirm, you won't hear from {{ mailinglist }} ever
again.</p >
  <p>Thanks,</p >

  <p>Your friendly internet Mail Ape  !</p>
{% endblock %}
```

Our template looks just like an HTML web page template, but it will be used in an email. Just like a normal Django template, we're extending a base template and filling out a block. In our case, the template we're extending is the `email/base.html` template we created at the start of this chapter. Also, note how we're using variables that we provided in our `send_confirmation_email()` function (for example, `confirmation_link`) and our `EmailTemplateContext` (for example, `mailing_list`).

Emails can include HTML but are not always rendered by web browsers. Notably, some versions of Microsoft Outlook use the Microsoft Word HTML renderer to render emails. Even Gmail, which runs in a browser, manipulates the HTML it receives before rendering it. Be careful to test complicated layouts in real email clients.

Next, let's create the plain text version of this template.

Creating the text confirmation email template

Now, we will create the plain text version of our confirmation email template; let's create it in `django/mailinglist/templates/mailinglist/email_templates/confirm_subscription.txt`:

```
Hello {{subscriber.email}},

Someone (hopefully you) just subscribed to {{ mailinglist }}.

To confirm your subscription go to {{confirmation_link}}.

If you don't confirm you won't hear from {{ mailinglist }} ever again.

Thanks,

Your friendly internet Mail Ape  !
```

In the preceding case, we're not using any HTML nor extending any base template.

However, we're still referencing variables that we provided in our `send_confirmation_email()` (for example, `confirmation_link`) function and our `EmailTemplateContext` class (for example, `mailing_list`).

Now that we have all the code necessary for sending emails, let's send them out when we create a new `Subscriber` model instance.

Sending on new Subscriber creation

As the final step, we'll take sending confirmation emails to users; we need to call our `send_confirmation_email` function. Based on the philosophy of fat models, we will call our `send_confirmation_email` function from our `Subscriber` model rather than a view. In our case, we will send the email when a new `Subscriber` model instance is saved.

Let's update our `Subscriber` model to send a confirmation email when a new `Subscriber` has been saved. To add this new behavior, we will need to edit `django/mailinglist/models.py`:

```
from django.db import models
from mailinglist import emails

class Subscriber(models.Model):
    # skipping unchanged model body

    def save(self, force_insert=False, force_update=False, using=None,
             update_fields=None):
        is_new = self._state.adding or force_insert
        super().save(force_insert=force_insert, force_update=force_update,
                     using=using, update_fields=update_fields)
        if is_new:
            self.send_confirmation_email()

    def send_confirmation_email(self):
           emails.send_confirmation_email(self)
```

The best way to add a new behavior when a model is created is to override the model's `save()` method. When overriding `save()`, it is vital that we still call the super class's `save()` method to make sure that the model does save. Our new save method does three things:

- Checks whether the current model is a new model
- Calls the super class's `save()` method
- Sends the confirmation email if the model is new

To check if the current model instance is new, we check the `_state` attribute. The `_state` attribute is an instance of the `ModelState` class. Generally, attributes that begin with an underscore (_) are considered private and may change across Django releases. However, the `ModelState` class is described in Django's official documentation so we can feel more comfortable using it (though we should keep an eye on future release notes for changes). If the `self._state.adding` is `True`, then the `save()` method is going to insert this model instance as a new row. If `self._state.adding` is `True`, then the `save()` method is going to update an existing row.

We've also wrapped the call to `emails.send_confirmation_email()` in a `Subscriber` method. This will be useful if we ever want to resend a confirmation email. Any code that wants to resend a confirmation email will not have to know about the `emails` module. The model is the expert on all its operations. This is the heart of the fat model philosophy.

A quick review of the section

In this section, we've learned more about Django's template system and how to send emails. We've learned how to render a template without using one of Django's built-in views to render it for us using the Django template engine directly. We've used the Django best practice of creating a service module to isolate all our email code. Finally, we've also used `send_email()` to send an email with a text and HTML body.

Next, let's use Celery to send these emails after we return a response to our users.

Using Celery to send emails

As we build increasingly complicated applications, we often want to perform operations without forcing the user to wait on us to return them an HTTP response. Django works well with Celery, a popular Python distributed task queue, to accomplish this.

Celery is a library to *queue tasks* in *brokers* to be processed by Celery *workers*. Let's take a closer look at some of these terms:

- A **Celery task** encapsulates a callable we want executed asynchronously.
- A **Celery queue** is a list of tasks in a first in, first out order stored in a broker.

- A **Celery broker** is a server that provides fast and efficient storage of queues. Popular brokers include RabbitMQ, Redis, and AWS SQS. Celery has different levels of support for different brokers. We will use Redis as our broker in development.
- **Celery workers** are separate processes that check queues for tasks to execute and execute them.

In this section, we will be doing the following things:

1. Installing Celery
2. Configuring Celery to work with Django
3. Using Celery queue a send confirmation email task
4. Using a Celery worker to send our emails

Let's start by installing Celery.

Installing celery

To install Celery, we'll update our `requirements.txt` file with these new changes:

```
celery<4.2
celery[redis]
django-celery-results<2.0
```

We will install three new packages and their dependencies:

- `celery`: Installs the main Celery package
- `celery[redis]`: Installs the dependencies we need to use Redis as our broker
- `django-celery-results`: Lets us store the results of executed tasks in our Django database; this is just one way of storing and logging Celery's results

Next, let's install our new packages using `pip`:

```
$ pip install -r requirements.txt
```

Now that we have Celery installed, let's configure Mail Ape to use Celery.

Configuring Celery settings

To configure Celery, we will need to make two sets of changes. First, we'll update the Django config to use Celery. Second, we'll create a celery configuration file that our worker will use.

Let's start by updating `django/config/settings.py`:

```
INSTALLED_APPS = [
    'user',
    'mailinglist',

    'crispy_forms',
    'markdownify',
    'django_celery_results',

    'django.contrib.admin',
    # other built in django apps unchanged.
]

CELERY_BROKER_URL = 'redis://localhost:6379/0'
CELERY_RESULT_BACKEND = 'django-db'
```

Let's take a closer look at these new settings:

- `django_celery_results`: This is a Celery extension that we installed as a Django app to let us store the results of our Celery tasks in the Django DB.
- `CELERY_BROKER_URL`: This is the URL to our Celery broker. In our case, we will use a local Redis server in development.
- `CELERY_RESULT_BACKEND`: This indicates where to store the results. In our case, we will use the Django database.

Since the `django_celery_results` app lets us save results in the database, it includes new Django models. For those models to exist in the database, we will need to migrate our database:

```
$ cd django
$ python manage.py migrate django_celery_results
```

Next, let's create a configuration file for our Celery worker. The worker will need an access to Django and our Celery broker.

Let's create the Celery worker configuration in `django/config/celery.py`:

```
import os
from celery import Celery

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'config.settings')

app = Celery('mailape')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

Celery knows how to work with a Django project out of the box. Here, we configure an instance of the Celery library based on our Django configuration. Let's review these settings in detail:

- `setdefault('DJANGO_SETTINGS_MODULE', ...)`: This ensures that our Celery worker knows which Django settings module to use if the `DJANGO_SETTINGS_MODULE` environment variable is not set for it.
- `Celery('mailape')`: This instantiates the Celery library for Mail Ape. Most Django apps use only one Celery instance, so the `mailape` string is not significant.
- `app.config_from_object('django.conf:settings', namespace='CELERY')`: This tells our Celery library to configure itself from the object at `django.conf.settings`. The `namespace` argument tells Celery that its settings are prefixed with `CELERY`.
- `app.autodiscover_tasks()`: This lets us avoid registering tasks by hand. When Celery is working with Django, it will check each installed app for a `tasks` module. Any tasks in that module will be automatically discovered.

Let's learn more about tasks by creating a task to send confirmation emails.

Creating a task to send confirmation emails

Now that Celery is configured, let's create a task to send a confirmation email to a subscriber.

A Celery task is a subclass of `Celery.app.task.Task`. However, most of the time when we create Celery tasks, we use Celery's decorators to mark a function as a task. In a Django project, it's often simplest to use the `shared_task` decorator.

When creating a task, it's useful to think of it like a view. The Django community's best practices recommend *thin views*, which means that views should be simple. They should not be responsible for complicated tasks, but should delegate that work to the model or a service module (for example, our `mailinglist.emails` module).

Keep task functions simple and put all the logic in models or service modules.

Let's create a task to send our confirmation emails in `django/mailinglist/tasks.py`:

```
from celery import shared_task

from mailinglist import emails


@shared_task
def send_confirmation_email_to_subscriber(subscriber_id):
    from mailinglist.models import Subscriber
    subscriber = Subscriber.objects.get(id=subscriber_id)
    emails.send_confirmation_email(subscriber)
```

There are a few unique things about our `send_confirmation_email_to_subscriber` function:

- `@shared_task`: This is a Celery decorator that turns a function into a `Task`. A `shared_task` is available to all Celery instances (in most Django cases, there's only one anyway).
- `def send_confirmation_email_to_subscriber(subscriber_id)`: This is a regular function that takes a subscriber ID as an argument. A Celery task can receive any pickle-able object (including a Django model). However, if you're passing around something that may be viewed as confidential (for example, an email address), you may wish to limit the number of systems that store the data (for example, not store it on the broker). In this case, we're passing our task function an ID of the `Subscriber` instead of the full `Subscriber`. The task function then queries the database for the related `Subscriber` instance.

A final item of note in this function is that we import the `Subscriber` model inside the function instead of at the top of the file. In our case, we will have our `Subscriber` model call this task. If we import the `models` module at the top of `tasks.py` and import the `tasks` module at the top of `model.py`, then we'll have a cyclic import error. In order to prevent that, we import `Subscriber` inside the function.

Next, let's call our task from `Subscriber.send_confirmation_email()`.

Sending emails to new subscribers

Now that we have our task, let's update our `Subscriber` to send confirmation emails using the task instead of using the `emails` module directly.

Let's update `django/mailinglist/models.py`:

```
from django.db import models
from mailinglist import tasks

class Subscriber(models.Model):
    # skipping unchanged model

     def send_confirmation_email(self):
        tasks.send_confirmation_email_to_subscriber.delay(self.id)
```

In our updated `send_confirmation_email()` method, we will take a look at how to call a task asynchronously.

A Celery task can be called either synchronously or asynchronously. Using the regular `()` operator, we'll call the task synchronously (for example, `tasks.send_confirmation_email_to_subscriber(self.id)`). A task that executes synchronously executes like a regular function call.

A Celery task also has the `delay()` method to execute a task asynchronously. When a task is told to execute asynchronously, it will queue a message in Celery's message broker. The Celery workers will then (eventually) pull the message from the broker's queue and execute the task. The result of the task is stored in the storage backend (in our case, the Django database).

Calling a task asynchronously returns a `result` object that offers a `get()` method. Calling `result.get()` blocks the current thread until the task has finished. `result.get()` then returns the result of the task. In our case, our tasks will not return anything, so we won't use the `result` function..

`task.delay(1, a='b')` is actually a shortcut for `task.apply_async((1,), kwargs={'a':'b'})`. Most of the time, the shortcut method is what we want. If you need a greater degree of control over your tasks execution, `apply_async()` is documented in the Celery documentation (`http://docs.celeryproject.org/en/latest/userguide/calling.html`).

Now that we can call tasks, let's start a worker to process our queued tasks.

Starting a Celery worker

Starting a Celery worker does not require us to write any new code. We can start one from the command line:

```
$ cd django
$ celery worker -A config.celery -l info
```

Let's look at all the arguments we gave `celery`:

- `worker`: This indicates that we want to start a new worker.
- `-A config.celery`: This is the app, or configuration, we want to use. In our case, the app we want is configured in `config.celery`.
- `-l info`: This is the log level to output. In this case, we're using `info`. By default, the level is `WARNING`.

Our worker is now able to process tasks queued by our code in Django. If we find we're queueing a lot of tasks, we can just start more `celery worker` processes.

A quick review of the section

In this section, you learned how to use Celery to process tasks asynchronously.

We learned how to set the broker and backend using the `CELERY_BROKER_URL` and `CELERY_RESULT_BACKEND` settings in our `settings.py`. We also created a `celery.py` file for our celery worker. Then, we used the `@shared_task` decorator to make a function a Celery task. With the task available, we learned how to call a Celery task with the `.delay()` shortcut method. Finally, we started a Celery worker to execute queued tasks.

Now that we know the basics, let's use this approach to send messages to our subscribers.

Sending messages to subscribers

In this section, we're going to create the `Message` model instances that represent messages that our users want to send to their mailing lists.

To send these messages, we will need to do the following things:

- Create a `SubscriberMessage` model to track which messages got sent and when
- Create a `SubscriberMessage` model instance for each confirmed `Subscriber` model instance associated with the new `Message` model instance
- Have `SubscriberMessage` model instances send an email to their associated `Subscriber` model instance's email

To make sure that even a `MailingList` model instance with lots of related `Subscriber` model instances doesn't slow down our website, we will use Celery to build our list of `SubscriberMessage` model instances *and* send the emails.

Let's start by creating a `SubscriberManager` to help us get a list of confirmed `Subscriber` model instances.

Getting confirmed subscribers

Good Django projects use custom model managers to centralize and document `QuerySet` objects related to their models. We need a `QuerySet` object to retrieve all the confirmed `Subscriber` model instances that belong to a given `MailingList` model instance.

Let's update `django/mailinglist/models.py` to add a new `SubscriberManager` class that knows how to get confirmed `Subscriber` model instances for a `MailingList` model instance:

```
class SubscriberManager(models.Manager):

    def confirmed_subscribers_for_mailing_list(self, mailing_list):
        qs = self.get_queryset()
        qs = qs.filter(confirmed=True)
        qs = qs.filter(mailing_list=mailing_list)
        return qs


class Subscriber(models.Model):
    # skipped fields
```

```
    objects = SubscriberManager()

    class Meta:
        unique_together = ['email', 'mailing_list', ]

    # skipped methods
```

Our new `SubscriberManager` object replaces the default manager in `Subscriber.objects`. The `SubscriberManager` class offers the `confirmed_subscribers_for_mailing_list()` method as well as all the methods of the default manager.

Next, let's create the `SubscriberMessage` model.

Creating the SubscriberMessage model

Now, we will create a `SubscriberMessage` model and manager. The `SubscriberMessage` model will let us track whether we successfully sent an email to a `Subscriber` model instance. The custom manager will have a method of creating all the `SubscriberMessage` model instances that a `Message` model instance needs.

Let's start by creating our `SubscriberMessage` in `django/mailinglist/models.py`:

```
import uuid

from django.conf import settings
from django.db import models

from mailinglist import tasks

class SubscriberMessage(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    message = models.ForeignKey(to=Message, on_delete=models.CASCADE)
    subscriber = models.ForeignKey(to=Subscriber, on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True)
    sent = models.DateTimeField(default=None, null=True)
    last_attempt = models.DateTimeField(default=None, null=True)

    objects = SubscriberMessageManager()

    def save(self, force_insert=False, force_update=False, using=None,
             update_fields=None):
        is_new = self._state.adding or force_insert
```

```
        super().save(force_insert=force_insert, force_update=force_update,
using=using,
             update_fields=update_fields)
        if is_new:
            self.send()

    def send(self):
        tasks.send_subscriber_message.delay(self.id)
```

Our `SubscriberMessage` model is pretty heavily customized compared to most of our other models:

- The `SubsriberMessage` fields connect it to a `Message` and a `Subscriber` let it track when it was created, last tried to send an email, and succeeded.
- `SubscriberMessage.objects` is a custom manager that we'll create in the following section.
- `SubscriberMessage.save()` works similar to `Subscriber.save()`. It checks whether the `SubscriberMessage` is new and whether it calls the `send()` method.
- `SubscriberMessage.send()` queues a task to send the message. We'll create that task later in the *Sending emails to Subscribers* section.

Now, let's create a `SubscriberMessageManager` in `django/mailinglist/models.py`:

```
from django.db import models


class SubscriberMessageManager(models.Manager):

    def create_from_message(self, message):
        confirmed_subs = Subscriber.objects.\
            confirmed_subscribers_for_mailing_list(message.mailing_list)
        return [
            self.create(message=message, subscriber=subscriber)
            for subscriber in confirmed_subs
        ]
```

Our new manager offers a method of creating `SubscriberMessages` from a `Message`. The `create_from_message()` method returns a list of `SubscriberMessages` each created using the `Manager.create()` method.

Finally, in order to have a new model available, we will need to create a migration and apply it:

```
$ cd django
$ python manage.py makemigrations mailinglist
$ python manage.py migrate mailinglist
```

Now that we have our `SubscriberMessage` model and table, let's update our project to automatically create `SubscriberMessage` model instances when a new `Message` is created.

Creating SubscriberMessages when a message is created

Mail Ape is meant to send a message as soon as it is created. For a `Message` model instance to become an email in a subscriber's inbox, we will need to build a set of `SubscriberMessage` model instances. The best time to build that set of `SubscriberMessage` model instances is just after a new `Message` model instance is created.

Let's override `Message.save()` in `django/mailinglist/models.py`:

```
class Message(models.Model):
    # skipped fields

    def save(self, force_insert=False, force_update=False, using=None,
             update_fields=None):
        is_new = self._state.adding or force_insert
        super().save(force_insert=force_insert, force_update=force_update,
                     using=using, update_fields=update_fields)
        if is_new:
            tasks.build_subscriber_messages_for_message.delay(self.id)
```

Our new `Message.save()` method follows a similar pattern as before. `Message.save()` checks whether the current `Message` is new and whether it then queues the `build_subscriber_messages_for_message` task for execution.

We'll use Celery to build the set of `SubscriberMessage` model instances asynchronously because we don't know how many `Subscriber` model instances are related to our `MailingList` model instance. If there are very many related `Subscriber` model instances, then it might make our web server unresponsive. Using Celery, our web server will return a response as soon as the `Message` model instance is saved. The `SubscriberMessage` model instances will be created by an entirely separate process.

Let's create the `build_subscriber_messages_for_message` task in `django/mailinglist/tasks.py`:

```
from celery import shared_task


@shared_task
def build_subscriber_messages_for_message(message_id):
    from mailinglist.models import Message, SubscriberMessage
    message = Message.objects.get(id=message_id)
    SubscriberMessage.objects.create_from_message(message)
```

As we discussed previously, our task doesn't contain much logic in itself. `build_subscriber_messages_for_message` lets the `SubscriberMessage` manager encapsulate all the logic of creating the `SubscriberMessage` model instances.

Next, let's write the code for sending emails that contain the `Message` our users create.

Sending emails to subscribers

Our final step in this section will be to send an email based on a `SubscriberMessage`. Earlier, we had our `SubscriberMessage.save()` method queue a task to send a `Subscriber` a `Message`. Now, we'll create that task and update the `emails.py` code to send the emails.

Lets's start by updating `django/mailinglist/tasks.py` with a new task:

```
from celery import shared_task


@shared_task
def send_subscriber_message(subscriber_message_id):
    from mailinglist.models import SubscriberMessage
    subscriber_message = SubscriberMessage.objects.get(
        id=subscriber_message_id)
    emails.send_subscriber_message(subscriber_message)
```

This new task follows the same pattern as the previous tasks we've created:

- We use the `shared_task` decorator to turn a regular function into a Celery task
- We import our model inside our task function to prevent a cyclical import error
- We let the `emails` module do the actual work of sending the email

Next, let's update the `django/mailinglist/emails.py` file to send emails based on a `SubscriberMessage`:

```
from datetime import datetime

from django.conf import settings
from django.core.mail import send_mail
from django.template import engines
from django.utils.datetime_safe import datetime

SUBSCRIBER_MESSAGE_TXT = 'mailinglist/email/subscriber_message.txt'

SUBSCRIBER_MESSAGE_HTML = 'mailinglist/email/subscriber_message.html'


def send_subscriber_message(subscriber_message):
    message = subscriber_message.message
    context = EmailTemplateContext(subscriber_message.subscriber, {
        'body': message.body,
    })

    dt_engine = engines['django'].engine
    text_body_template = dt_engine.get_template(SUBSCRIBER_MESSAGE_TXT)
    text_body = text_body_template.render(context=context)
    html_body_template = dt_engine.get_template(SUBSCRIBER_MESSAGE_HTML)
    html_body = html_body_template.render(context=context)

    utcnow = datetime.utcnow()
```

```
    subscriber_message.last_attempt = utcnow
    subscriber_message.save()

    success = send_mail(
        subject=message.subject,
        message=text_body,
        from_email=settings.MAILING_LIST_FROM_EMAIL,
        recipient_list=(subscriber_message.subscriber.email,),
        html_message=html_body)

    if success == 1:
        subscriber_message.sent = utcnow
        subscriber_message.save()
```

Our new function takes the following steps:

1. Builds the context for the templates using the `EmailTemplateContext` class we created earlier
2. Renders the text and HTML versions of the email using the Django Template engine
3. Records the time of the current sending attempt
4. Sends the email using Django's `send_mail()` function
5. If `send_mail()` returned that it sent an email, it records the time the message was sent

Our `send_subscriber_message()` function requires us to create HTML and text templates to render.

Let's create our HTML email body template in `django/mailinglist/templates/mailinglist/email_templates/subscriber_message.html`:

```
{% extends "mailinglist/email_templates/email_base.html" %}
{% load markdownify %}

{% block body %}
  {{ body | markdownify }}
{% endblock %}
```

This template renders the markdown body of the `Message` into HTML. We've used the `markdownify` tag library to render markdown into HTML before. We don't need HTML boilerplate or to include an unsubscribe link footer because the `email_base.html` already does that.

Next, we must create the text version of the message template in `mailinglist/templates/mailinglist/email_templates/subscriber_message.txt`:

```
{{ body }}

---

You're receiving this message because you previously subscribed to {{
mailinglist }}.

If you'd like to unsubsribe go to {{ unsubscription_link }} and click
unsubscribe.

Sent with Mail Ape  .
```

This template looks very similar. In this case, we simply output the body as un-rendered markdown. Also, we don't have a base template for our text emails, so we have to write out the footer with an unsubscribe link manually.

Congratulations! You've now updated Mail Ape to send emails to mailing list subscribers.

Make sure that you restart your `celery worker` process(es) any time you change your code. `celery worker` does not include an automatic restart like the Django `runserver`. If we don't restart the `worker`, then it won't get any updated code changes.

Next, let's make sure that we can run our tests without triggering Celery or sending an actual email.

Testing code that uses Celery tasks

At this point, two of our models will automatically queue Celery tasks when they are created. This can create a problem for us when testing our code since we may not want to have a Celery broker running when we run our tests. Instead, we should use Python's `mock` library to prevent the need for an outside system to be running when we run our tests.

One approach we could use is to decorate each test method that uses the `Subscriber` or `Message` models with Python's `@patch()` decorator. However, this manual process is likely to be error-prone. Let's look at some alternatives instead.

In this section, we will take a look at two approaches to make mocking out Celery tasks easier:

- Using a mixin to prevent the `send_confirmation_email_to_subscriber` task from being queued in any test
- Using a Factory to prevent the `send_confirmation_email_to_subscriber` task from being queued

By fixing the same problem in two different ways, you'll get insight into which solution works better in which situation. You may find that having both options available in a project is helpful.

We can use the exact same approaches for patching references to `send_mail` to prevent emails being sent out during testing.

Let's start by using a mixin to apply a patch.

Using a TestCase mixin to patch tasks

In this approach, we will create a mixin that `TestCase` authors can optionally use when writing `TestCase`s. We've used mixins in a lot of our Django code to override the behavior of class-based views. Now, we'll create a mixin that will override the default behavior of `TestCase`s. We will take advantage of each test method being preceded by a call to `setUp()` and followed by `tearDown()` to set up our patch and mock.

Let's create our mixin `django/mailinglist/tests.py`:

```
from unittest.mock import patch


class MockSendEmailToSubscriberTask:

    def setUp(self):
        self.send_confirmation_email_patch = patch(
            'mailinglist.tasks.send_confirmation_email_to_subscriber')
        self.send_confirmation_email_mock =
```

```
        self.send_confirmation_email_patch.start()
        super().setUp()

    def tearDown(self):
        self.send_confirmation_email_patch.stop()
        self.send_confirmation_email_mock = None
        super().tearDown()
```

Our mixin's `setUp()` method does three things:

- Creates a patch and saves it as an attribute of our object
- Starts the patch and saves the resulting mock object as an attribute of our object Access to the mock is important so that we can later assert what it was called
- Calls the parent class's `setUp()` method so that the `TestCase` is properly set up

Our mixin's `tearDown` method also does the following three things:

- Stops the patch
- Removes a reference to the mock
- Calls the parent class's `tearDown` method to complete any other cleanup that needs to happen

Let's create a `TestCase` to test `SubscriberCreation` and take a look at our new `MockSendEmailToSubscriberTask` in action. We'll create a test that creates a `Subscriber` model instance using its manager's `create()` method. The `create()` call will in turn call `save()` on the new `Subscriber` instances. The `Subscriber.save()` method should then queue a `send_confirmation_email` task.

Let's add our test to `django/mailinglist/tests.py`:

```
from mailinglist.models import Subscriber, MailingList

from django.contrib.auth import get_user_model
from django.test import TestCase


class SubscriberCreationTestCase(
    MockSendEmailToSubscriberTask,
    TestCase):

    def test_calling_create_queues_confirmation_email_task(self):
        user = get_user_model().objects.create_user(
            username='unit test runner'
        )
```

```
            mailing_list = MailingList.objects.create(
                name='unit test',
                owner=user,
            )
            Subscriber.objects.create(
                email='unittest@example.com',
                mailing_list=mailing_list)
    self.assertEqual(self.send_confirmation_email_mock.delay.call_count, 1)
```

Our test asserts that the mock we created in our mixin has been called once. This gives us confidence that when we create a new `Subscriber`, we will queue the correct task.

Next, let's look at how we can solve this problem using Factory Boy factories.

Using patch with factories

We discussed using Factory Boy factories in `Chapter 8`, *Testing Answerly*. Factories make it easier to create complicated objects. We will now take a look at how to use Factories and Python's `patch()` together to prevent tasks from being queued.

Let's create a `SubscriberFactory` in `django/mailinglist/factories.py`:

```
from unittest.mock import patch

import factory

from mailinglist.models import Subscriber


class SubscriberFactory(factory.DjangoModelFactory):
    email = factory.Sequence(lambda n: 'foo.%d@example.com' % n)

    class Meta:
        model = Subscriber

    @classmethod
    def _create(cls, model_class, *args, **kwargs):
        with
patch('mailinglist.models.tasks.send_confirmation_email_to_subscriber'):
            return super()._create(model_class=model_class, *args,
**kwargs)
```

Our factory overrides the default `_create()` method to apply the task patch before the default `_create()` method is called. When the default `_create()` method executes, it will call `Subscriber.save()`, which will try to queue the `send_confirmation_email` task. However, the task will be replaced with a mock. Once the model is created and the `_create()` method returns, the patch will be removed.

We can now use our `SubscriberFactory` in a test. Let's write a test in `django/mailinglist/tests.py` to verify that `SubscriberManager.confirmed_subscribers_for_mailing_list()` works correctly:

```
from django.contrib.auth import get_user_model
from django.test import TestCase

from mailinglist.factories import SubscriberFactory
from mailinglist.models import Subscriber, MailingList


class SubscriberManagerTestCase(TestCase):

    def testConfirmedSubscribersForMailingList(self):
        mailing_list = MailingList.objects.create(
            name='unit test',
            owner=get_user_model().objects.create_user(
                username='unit test')
        )
        confirmed_users = [
            SubscriberFactory(confirmed=True, mailing_list=mailing_list)
            for n in range(3)]
        unconfirmed_users = [
            SubscriberFactory(mailing_list=mailing_list)
            for n in range(3)]
        confirmed_users_qs =
Subscriber.objects.confirmed_subscribers_for_mailing_list(
            mailing_list=mailing_list)
        self.assertEqual(len(confirmed_users), confirmed_users_qs.count())
        for user in confirmed_users_qs:
            self.assertIn(user, confirmed_users)
```

Now that we've seen both approaches, let's look at some of the trade-offs between the two approaches.

Choosing between patching strategies

Both Factory Boy factories and `TestCase` mixins help us solve the problem of how to test code that queues a Celery task without queuing a Celery task. Let's take a closer look at some of the trade-offs.

Some of the trade-offs when using a mixin are as follows:

- The patch stays in place during the entire test
- We have access to the resulting mock
- The patch will be applied even on tests that don't need it
- The mixins in our `TestCase` are dictated by the models we reference in our code, which can be a confusing level of indirection for test authors

Some of the trade-offs when using a Factory are as follows:

- We can still access the underlying function in a test if we need to.
- We don't have access to the resulting mock to assert (we often don't need it).
- We don't connect the parent class of `TestCase` to the models we're referring to in our test methods. It's simpler for test authors.

The ultimate decision for which approach to use is dictated by the test we're writing.

Summary

In this chapter, we gave Mail Ape the ability to send emails to our users' `MailingList`'s confirmed `Subscribers`. We also learned how to use Celery to process tasks outside of Django's request/response cycle. This lets us process tasks that may take a long time or require other resources (for example, SMTP servers and more memory) without slowing down our Django web servers.

We covered a variety of email and Celery-related topics in this chapter. We saw how to configure Django to use an SMTP server. We used Django's `send_email()` function to send emails. We created a Celery task with the `@shared_task` decorator. We queued a Celery task using its `delay()` method. Finally, we explored some useful approaches for testing code that relies on external resources.

Next, let's build an API for our Mail Ape so that our users can integrate into their own websites and apps.

12
Building an API

Now that Mail Ape can send emails to our subscribers, let's make it easier for our users to integrate with Mail Ape using an API. In this chapter, we will build a RESTful JSON API that will let users create mailing lists and add subscribers to a mailing list. To simplify creating our API, we will use the **Django REST framework** (**DRF**). Finally, we'll access our API on the command line using curl.

In this chapter, we will do the following things:

- Summarize the core concepts of DRF
- Create `Serializer`s that define how to parse and serialize `MailingList` and `Subscriber` models
- Create a permission class to restrict API to users who are`MailingList` owners
- Use the Django REST framework's class-based views to create the views for our API
- Access our API over HTTP using curl
- Test our API in a unit test

Let's start this chapter with DRF.

Starting with the Django REST framework

We'll start by installing DRF and then reviewing its configuration. As we review the DRF configuration, we'll learn about the features and concepts that make it useful.

Installing the Django REST framework

Let's start by adding DRF to our `requirements.txt` file:

```
djangorestframework<3.8
```

Next, we can install it using `pip`:

```
$ pip install -r requirements.txt
```

Now that we have the library installed, let's add DRF to our `INSTALLED_APPS` list in the `django/mailinglist/settings.py` file:

```
INSTALLED_APPS = [
# previously unchanged list
    'rest_framework',
]
```

Configuring the Django REST Framework

DRF is highly configurable through its view classes. However, we can avoid repeating the same common settings across all our DRF views using DRF's settings in our `settings.py` file.

All of DRF's features project out from how DRF handles views. DRF provides a rich collection of views that extend `APIView` (which in turn extends Django's `View` class). Let's look at the APIView's life cycle and the related settings.

A DRF view's life cycle perform the following actions:

1. **Wrap Django's request object in the DRF request object**: DRF has a specialized `Request` class that wraps Django's `Request` class, as will be discussed in the following sections.
2. **Perform content negotiation**: Find the request parser and response renderer.
3. **Perform authentication**: Check the credentials associated with the request.
4. **Check permissions**: Checks whether the user associated with the request can access this view.
5. **Check throttles**: Checks whether there haven't been too many requests recently by this user.
6. **Execute the view handler**: Performs the action associated with the view (for example, creating the resource, querying the database, and so on).
7. **Render the response**: Renders the response to the correct content type.

DRF's custom `Request` class is much like Django's `Request` class, except that it can be configured with a parser. A DRF view finds the correct parser for the request based on the view's settings and the content type of the request during content negotiation. The parsed contents are available as `request.data` just like a Django request with a `POST` form submission.

DRF views also use a specialized `Response` class that uses a render instead of a Django template. The renderer is selected during the content negotiation step.

Most of the preceding steps are performed using configurable classes. DRF is configurable by creating a dictionary in our project's `settings.py` under the name `REST_FRAMEWORK`. Let's review some of the most important settings:

- `DEFAULT_PARSER_CLASSES`: This supports JSON, forms and multipart forms by default. Other parsers (for example, YAML and MessageBuffer) are available as third-party community packages.
- `DEFAULT_AUTHENTICATION_CLASSES`: This supports session-based authentication and HTTP basic authentication by default. Session authentication can make using your API in your app's frontend easier. DRF ships with a token authentication class. OAuth (1 and 2) support is available through third-party community packages.
- `DEFAULT_PERMISSION_CLASSES`: This defaults to allowing any user to any action (including update and delete operations). DRF ships with a collection of stricter permissions listed in the documentation (`https://www.django-rest-framework.org/api-guide/permissions/#api-reference`). We'll also take a look at how to create a custom permission class later in this chapter.
- `DEFAULT_THROTTLE_CLASSES`/`DEFAULT_THROTTLE_RATES`: This is empty (unthrottled) by default. DRF offers a simple throttling scheme, letting us set different rates for anonymous requests and user requests out of the box.
- `DEFAULT_RENDERER_CLASSES`: This defaults to JSON and a *browsable* template renderer. The browsable template renderer makes a simple UI for view and testing your views, suitable to development.

We will configure our DRF to be a bit stricter, even in development. Let's update `django/config/settings.py` with the following new setting `dict`:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.UserRateThrottle',
        'rest_framework.throttling.AnonRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'user': '60/minute',
        'anon': '30/minute',
    },
}
```

This configuration restricts the API to authenticated users by default and sets a throttle on their requests. Authenticated users can make 60 requests per minute before being throttled. Unauthenticated users can make 30 requests per minute. DRF accepts throttle periods of `second`, `minute`, `hour`, or `day`.

Next, let's take a look at DRF `Serializers`.

Creating the Django REST Framework Serializers

When a DRF parser parses a request's body, the parser basically returns a Python dictionary. However, before we can perform any operation on that data, we will need to confirm that the data is valid. In our previous Django views, we'd use a Django form. In DRF, we use a `Serializer` class.

DRF `Serializer` classes are very similar to Django form classes. Both are involved in receiving validation data and preparing models for output. However, a `Serializer` class doesn't know how to render its data, unlike a Django form that does. Remember that in a DRF view, the renderers are responsible for rendering the result into JSON or whatever format was negotiated by the request.

Much like a Django form, a `Serializer` can be created to work on an arbitrary data or be based on a Django Model. Also, `Serializer` is composed of a collection of fields that we can use to control serialization. When a `Serializer` is related to a model, the Django REST framework knows which serializer `Field` to use for which model `Field`, similar to how `ModelForm`s work.

Let's create a `Serializer` for our `MailingList` model in `django/mailinglist/serializers.py`:

```
from django.contrib.auth import get_user_model
from rest_framework import serializers

from mailinglist.models import MailingLIst


class MailingListSerializer(serializers.HyperlinkedModelSerializer):
    owner = serializers.PrimaryKeyRelatedField(
        queryset=get_user_model().objects.all())

    class Meta:
        model = MailingList
        fields = ('url', 'id', 'name', 'subscriber_set')
        read_only_fields = ('subscriber_set', )
        extra_kwargs = {
            'url': {'view_name': 'mailinglist:api-mailing-list-detail'},
            'subscriber_set': {'view_name': 'mailinglist:api-subscriber-
detail'},
        }
```

This seems very similar to how we wrote `ModelForm`s; let's take a closer look:

- `HyperlinkedModelSerializer`: This is the `Serializer` class that shows a hyperlink to any related model, so when it shows the related `Subscriber` model instances of a `MailingList`, it will show a link (URL) to that instance's detail view.
- `owner = serializers.PrimaryKeyRelatedField(...)`: This changes the field for serializing the model's `owner` field. The `PrimaryKeyRelatedField` returns the related object's primary key. This is useful when the related model doesn't have a serializer or a related API view (like the user model in Mail Ape).
- `model = MailingList`: Tells our `Serializer` which mode it's serializing

- `fields = ('url', 'id', ...)`: This lists the model's fields to serialize. The `HyperlinkedModelSerializer` includes an extra field `url`, which is the URL to the serialized model's detail view. Much like with a Django `ModelForm`, a `ModelSerializer` class (such as `HyperlinkedModelSerializer`) has a set of default serializer fields for each model field. In our case, we've decided to override how `owner` is represented (refer to the preceding point about the `owner` attribute).
- `read_only_fields = ('subscriber_set', )`: This concisely lists which fields may not be modified. In our case, this prevents users from tampering with the mailing list a `Subscriber` is in.
- `extra_kwargs`: This dictionary lets us provide extra arguments to each field's constructor without overriding the entire field. This is usually done to provide a `view_name` argument which is needed to look up the URL to a view.
- `'url': {'view_name': '...'},`: This provides the name of the `MailingList` API detail view.
- `'subscriber_set': {'view_name': '...'},`: This provides the name of the `Subscriber` API detail view.

There are actually two ways of marking a `Serializer`'s field read only. One way is using the `read_only_fields` attribute as in the preceding code sample. Another is to pass `read_only=True` as an argument to a `Field` class's constructor (for example, `email = serializers.EmailField(max_length=240, read_only=True)`).

Next, we'll create two serializers for our `Subscriber` model. Our two subscribers are going to have one difference: whether `Subscriber.email` is editable. We will need to let users write to `Subscriber.email` when they're creating `Subscriber`. However, we don't want them to be able to change the email after they've created the user.

First, let's create a `Serializer` for the `Subscription` model in `django/mailinglist/serialiers.py`:

```
from rest_framework import serializers

from mailinglist.models import Subscriber


class SubscriberSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Subscriber
```

```
        fields = ('url', 'id', 'email', 'confirmed', 'mailing_list')
        extra_kwargs = {
            'url': {'view_name': 'mailinglist:api-subscriber-detail'},
            'mailing_list': {'view_name': 'mailinglist:api-mailing-list-
detail'},
        }
```

The `SubscriberSerializer` is just like our `MailingListSerializer`. We use many of the same elements:

- Subclassing `serializers.HyperlinkedModelSerializer`
- Declaring the related model using an inner `Meta` class's `model` attribute
- Declaring the related model's fields using an inner `Meta` class's `fields` attribute
- Giving the related model's detail view's name using the `extra_kwargs` dictionary and the `view_name` key.

For our next `Serializer` class, we'll create one just like `SubscriberSerializer` but make the `email` field read only; let's add it to `django/mailinglist/serialiers.py`:

```
from rest_framework import serializers

from mailinglist.models import Subscriber


class
ReadOnlyEmailSubscriberSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Subscriber
        fields = ('url', 'id', 'email', 'confirmed', 'mailing_list')
        read_only_fields = ('email', 'mailing_list',)
        extra_kwargs = {
            'url': {'view_name': 'mailinglist:api-subscriber-detail'},
            'mailing_list': {'view_name': 'mailinglist:api-mailing-list-
detail'},
        }
```

This `Serializer` lets us update whether a `Subscriber` is `confirmed` or not, but it won't let the `Subscriber`'s `email` field change.

Now that we've create a few `Serializers`, we can see how similar they are to Django's built-in `ModelForm`s. Next, let's create a `Permission` class to prevent a user from accessing each other's `MailingList` and `Subscriber` model instances.

API permissions

In this section, we'll create a permission class that the Django REST framework will use to check whether a user may perform an operation on a `MailingList` or `Subscriber`. This will perform a very similar role to the `UserCanUseMailingList` mixin we created in `Chapter 10`, Starting Mail Ape.

Let's create our `CanUseMailingList` class in `django/mailinglist/permissions.py`:

```
from rest_framework.permissions import BasePermission

from mailinglist.models import Subscriber, MailingList


class CanUseMailingList(BasePermission):

    message = 'User does not have access to this resource.'

    def has_object_permission(self, request, view, obj):
        user = request.user
        if isinstance(obj, Subscriber):
            return obj.mailing_list.user_can_use_mailing_list(user)
        elif isinstance(obj, MailingList):
            return obj.user_can_use_mailing_list(user)
        return False
```

Let's take a closer look at some of the new elements introduced in our `CanUseMailingList` class:

- `BasePermission`: Provides the basic contract of a permission class, implementing the `has_permission()` and `has_object_permission()` methods to always return `True`
- `message`: This is the message that the `403` response body
- `def has_object_permission(...)`: Checks whether the request's user is the owner of the related `MailingList`

The `CanUseMailingList` class doesn't override `BasePermission.has_permission(self, request, view)` because the permissions in our system are both at the object level rather than the view or model level.

If you need a more dynamic permission system, you may want to use Django's built-in permission system (`https://docs.djangoproject.com/en/2.0/topics/auth/default/#permissions-and-authorization`) or Django Guardian (`https://github.com/django-guardian/django-guardian`).

Now that we have our `Serializer`s and permission class, we will write our API views.

Creating our API views

In this section, we'll create the actual views that define Mail Ape's RESTful API. The Django REST framework offers a collection of class-based views that are similar to Django's suite of class-based views. One of the main differences between the DRF generic views and the Django generic views is how they combine multiple operations in a single view class. For example, DRF offers the `ListCreateAPIView` class but Django only offers a `ListView` class and a `CreateView` class. DRF offers a `ListCreateAPIView` class because a resource at `/api/v1/mailinglists` would be expected to provide both a list of `MailingList` model instances and a creation endpoint.

Django REST Framework also offers a suite of function decorators (`http://www.django-rest-framework.org/api-guide/views/#function-based-views`) so that you use function-based views too.

Let's learn more about DRF views by creating our API, starting with the `MailingList` API views.

Creating MailingList API views

Mail Ape will provide an API to create, read, update, and delete `MailingList`s. To support these operations, we will create the following two views:

- A `MailingListCreateListView` that extends `ListCreateAPIView`
- A `MailingListRetrieveUpdateDestroyView` that extends `RetrieveUpdateDestroyAPIView`

Listing MailingLists by API

To support getting a list of a user's `MailingList` model instances and creating new `MailingList` model instances, we will create the `MailingListCreateListView` class in `django/mailinglist/views.py`:

```
from rest_framework import generics
from rest_framework.permissions import IsAuthenticated

from mailinglist.permissions import CanUseMailingList
from mailinglist.serializers import MailingListSerializer


class MailingListCreateListView(generics.ListCreateAPIView):
    permission_classes = (IsAuthenticated, CanUseMailingList)
    serializer_class = MailingListSerializer

    def get_queryset(self):
        return self.request.user.mailinglist_set.all()

    def get_serializer(self, *args, **kwargs):
        if kwargs.get('data', None):
            data = kwargs.get('data', None)
            owner = {
                'owner': self.request.user.id,
            }
            data.update(owner)
        return super().get_serializer(*args, **kwargs)
```

Let's review our `MailingListCreateListView` class in detail:

- `ListCreateAPIView`: This is the DRF generic view we extend. It responds to `GET` requests with the serialized contents returned by the `get_queryset()` method. When it receives a `POST` request, it will create and return a `MailingList` model instance.
- `permission_classes`: This is a collection of permission classes that will be called in an order. If `IsAuthenticated` fails, then `IsOwnerPermission` won't be called.
- `serializer_class = MailingListSerializer`: This is the serializer this view uses.
- `def get_queryset(self)`: This is used to get a `QuerySet` of models to serialize and return.

- `def get_serializer(...)`: This is used to get the serializer instance. In our case, we're overriding the owner (if any) that we received as input from the request with the currently logged in user. By doing so, we ensure that a user can't create a mailing list owned by another. This is very similar to how we might override `get_initial()` in a Django form view (for example, refer to the `CreateMessageView` class from `Chapter 10`, *Starting Mail Ape*).

Now that we have our view, let's add it to our URLConf in `django/mailinglist/urls.py` with the following code:

```
    path('api/v1/mailing-list', views.MailingListCreateListView.as_view(),
         name='api-mailing-list-list'),
```

Now, we can create and list `MailingList` model instances by sending a request to `/mailinglist/api/v1/mailing-list`.

Editing a mailing list via an API

Next, let's add a view to view, update, and delete a single `MailingList` model instance by adding a new view to `django/mailinglist/views.py`:

```
from rest_framework import generics
from rest_framework.permissions import IsAuthenticated

from mailinglist.permissions import CanUseMailingList
from mailinglist.serializers import MailingListSerializer
from mailinglist.models import MailingList


class MailingListRetrieveUpdateDestroyView(
    generics.RetrieveUpdateDestroyAPIView):

    permission_classes = (IsAuthenticated, CanUseMailingList)
    serializer_class = MailingListSerializer
    queryset = MailingList.objects.all()
```

`MailingListRetrieveUpdateDestroyView` looks very similar to our previous view but extends the `RetrieveUpdateDestroyAPIView` class. Like Django's built-in `DetailView`, `RetrieveUpdateDestroyAPIView` expects that it will receive the `pk` of the `MailingList` model instance it is to operate on in the request's path. `RetrieveUpdateDestroyAPIView` knows how to handle a variety of HTTP methods:

- On a `GET` request, it retrieves the model identified by the `pk` argument
- On a `PUT` request, it overwrites all the fields of the model identified by the `pk` with the fields received in the argument
- On a `PATCH` request, it overwrites only the fields received in the request
- On a `DELETE` request, it deletes the model identified by the `pk`

Any updates (whether by `PUT` or by `PATCH`) are validated by the `MailingListSerializer`.

Another difference is that we define a `queryset` attribute for the view (`MailingList.objects.all()`) instead of a `get_queryset()` method. We don't need to restrict our `QuerySet` dynamically because the `CanUseMailingList` class will protect us from users editing/viewing `MailingLists` they don't have permission to access.

Just like before, we now need to connect our view to our app's URLConf in `django/mailinglist/urls.py` with the following code:

```
    path('api/v1/mailinglist/<uuid:pk>',
         views.MailingListRetrieveUpdateDetroyView.as_view(),
         name='api-mailing-list-detail'),
```

Note that we parse the `<uuid:pk>` argument out of the request's path just like we do with some of Django's regular views that operate on a single model instance.

Now that we have our `MailingList` API, let's allow our users to manage `Subscribers` by API as well.

Creating a Subscriber API

In this section, we'll create an API to manage `Subscriber` model instances. This API will be powered by two views:

- `SubscriberListCreateView` to list and create `Subscriber` model instances

- `SubscriberRetrieveUpdateDestroyView` to retrieve, update, and delete a `Subscriber` model instance

Listing and Creating Subscribers API

`Subscriber` model instances have an interesting difference from `MailingList` model instances in that `Subscriber` model instances are not directly related to a user. To get a list of `Subscriber` model instances, we need to know which `MailingList` model instance we should query. `Subscriber` model instance creation faces the same question, so both these operations will have to receive a related `MailingList`'s `pk` to execute.

Let's start with our by creating our `SubscriberListCreateView` in `django/mailinglist/views.py`:

```
from rest_framework import generics
from rest_framework.permissions import IsAuthenticated

from mailinglist.permissions import CanUseMailingList
from mailinglist.serializers import SubscriberSerializer
from mailinglist.models import MailingList, Subscriber


class SubscriberListCreateView(generics.ListCreateAPIView):
    permission_classes = (IsAuthenticated, CanUseMailingList)
    serializer_class = SubscriberSerializer

    def get_queryset(self):
        mailing_list_pk = self.kwargs['mailing_list_pk']
        mailing_list = get_object_or_404(MailingList, id=mailing_list_pk)
        return mailing_list.subscriber_set.all()

    def get_serializer(self, *args, **kwargs):
        if kwargs.get('data'):
            data = kwargs.get('data')
            mailing_list = {
                'mailing_list': reverse(
                    'mailinglist:api-mailing-list-detail',
                    kwargs={'pk': self.kwargs['mailing_list_pk']})
            }
            data.update(mailing_list)
        return super().get_serializer(*args, **kwargs)
```

Our `SubscriberListCreateView` class has much in common with our `MailingListCreateListView` class, including the same base class and `permission_classes` attribute. Let's take a closer look at some of the differences:

- `serializer_class`: Uses a `SubscriberSerializer`.
- `get_queryset()`: Checks whether the related `MailingList` model instance identified in the URL exists before returning a `QuerySet` of all the related `Subscriber` model instances.
- `get_serializer()`: Ensures the new `Subscriber` is associated with the `MailingList` in the URL. We use the `reverse()` function to identify the associated `MailingList` model instance because the `SubscriberSerializer` class is inherits from the `HyperlinkedModelSerializer` class. `HyperlinkedModelSerializer` wants related models to be identified by a hyperlink or path (not a `pk`).

Next, we will add a `path()` object for our `SubscriberListCreateView` class to the URLConf in `django/mailinglist/urls.py`:

```
path('api/v1/mailinglist/<uuid:mailing_list_pk>/subscribers',
     views.SubscriberListCreateView.as_view(),
     name='api-subscriber-list'),
```

When adding a `path()` object for our `SubscriberListCreateView` class, we will need to ensure that we have a `mailing_list_pk` parameter. This lets `SubscriberListCreateView` know which `Subscriber` model instances to operate on.

Our users are now able to add `Subscriber` s to their `MailingList` via our RESTful API. Adding a user to our API will then trigger a confirmation email because `Subscriber.save()` will be called by our `SubscriberSerializer`. Our API doesn't need to know how to send emails because our *fat model* is the expert on the behavior of `Subscriber`.

However, this API does present a potential bug in Mail Ape. Our current API lets us add a `Subscriber` who has been already confirmed. However, our `Subscriber.save()` method will send a confirmation email to the email address of all new `Subscriber` model instances. This can lead to us spamming the already confirmed `Subscriber`s. To fix this bug, let's update `Subscriber.save` in `django/mailinglist/models.py`:

```
class Subscriber(models.Model):
    # skipping unchanged attributes and methods

    def save(self, force_insert=False, force_update=False, using=None,
```

```
            update_fields=None):
        is_new = self._state.adding or force_insert
        super().save(force_insert=force_insert, force_update=force_update,
                     using=using, update_fields=update_fields)
        if is_new and not self.confirmed:
            self.send_confirmation_email()
```

Now, we're only calling `self.send_confirmation_email()` if we're saving a new *and* unconfirmed `Subscriber` model instance.

Great! Now, let's create a view to retrieve, update, and delete a `Subscriber` model instance.

Updating subscribers via an API

Now, that we have created and list API operations for Subscriber model instances, we can create an API view for retrieving, updating, and deleting a single `Subscriber` model instance.

Let's add our view to `django/mailinglist/views.py`:

```
from rest_framework import generics
from rest_framework.permissions import IsAuthenticated

from mailinglist.permissions import CanUseMailingList
from mailinglist.serializers import ReadOnlyEmailSubscriberSerializer
from mailinglist.models import Subscriber


class SubscriberRetrieveUpdateDestroyView(
    generics.RetrieveUpdateDestroyAPIView):

    permission_classes = (IsAuthenticated, CanUseMailingList)
    serializer_class = ReadOnlyEmailSubscriberSerializer
    queryset = Subscriber.objects.all()
```

Our `SubscriberRetrieveUpdateDestroyView` is very similar to our `MailingListRetrieveUpdateDestroyView` view. Both inherit from the same `RetrieveUpdateDestroyAPIView` class to provide the core behavior in response to HTTP requests and use the same `permission_classes` list. `SubscriberRetrieveUpdateDestroyView` however has two differences:

- `serializer_class = ReadOnlyEmailSubscriberSerializer`: This is a different `Serializer`. In the case of updates, we don't want the user to be able to change email addresses.
- `queryset = Subscriber.objects.all()`: This is a `QuerySet` of all `Subscribers`. We don't need to restrict the `QuerySet` because the `CanUseMailingList` will prevent unauthorized access.

Next, let's make sure that we can route to it by adding it to the `urlpatterns` list in `django/mailinglist/urls.py`:

```
    path('api/v1/subscriber/<uuid:pk>',
         views.SubscriberRetrieveUpdateDestroyView.as_view(),
         name='api-subscriber-detail'),
```

Now that we have our views, let's try interacting with it on the command line.

Running our API

In this section, we'll run Mail Ape on the command line and interact with our API on the command line using `curl`, a popular command-line tool used for interacting with servers. In this section, we will perform the following functions:

- Creating a user on the command line
- Creating a mailing list on the command line
- Getting a list of `MailingList` s on the command line
- Creating a `Subscriber` on the command line
- Getting a list of `Subscriber` s on the command line

Let's start by creating our user using the Django `manage.py shell` command:

```
$ cd django
$ python manage.py shell
Python 3.6.3 (default)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
In [1]: from django.contrib.auth import get_user_model

In [2]: user = get_user_model().objects.create_user(username='user',
password='secret')
In [3]: user.id
2
```

If you've already registered a user using the web interface, you can use that user. Also, never use `secret` as your password in production.

Now that we have a user who we can use on the command line, let's start our local Django server:

```
$ cd django
$ python manage.py runserver
```

Now that our server is running, we can open a different shell and get a list of `MailingList`s for our user:

```
$ curl "http://localhost:8000/mailinglist/api/v1/mailing-list" \
     -u 'user:secret'
[]
```

Let's take a closer look at our command:

- `curl`: This is the tool we're using.
- `"http://... api/v1/mailing-list"`: This is the URL we're sending our request to.
- `-u 'user:secret'`: This is the basic authentication credentials. `curl` takes care of encoding these correctly for us.
- `[]`: This is an empty JSON list returned by the server. In our case, `user` doesn't have any `MailingList`s yet.

We get a JSON response because the Django REST framework is configured to use JSON rendering by default.

To create a `MailingList` for our user, we will need to send a `POST` request like this:

```
$ curl -X "POST" "http://localhost:8000/mailinglist/api/v1/mailing-list" \
     -H 'Content-Type: application/json; charset=utf-8' \
     -u 'user:secret' \
     -d $'{
  "name": "New List"
}'
{"url":"http://localhost:8000/mailinglist/api/v1/mailinglist/cd983e25-c6c8-
48fa-9afa-1fd5627de9f1","id":"cd983e25-
c6c8-48fa-9afa-1fd5627de9f1","name":"New
List","owner":2,"subscriber_set":[]}
```

This is a much longer command with a proportionately longer result. Let's take a look at each new argument:

- `-H 'Content-Type: application/json; charset=utf-8' \`: This adds a new HTTP `Content-Type` header to tell the server to parse the body as JSON.
- `-d $'{ ... }'`: This specifies the body of the request. In our case, we're sending a JSON object with the name of the new mailing list.
- `"url":"http://...cd983e25-c6c8-48fa-9afa-1fd5627de9f1"`: This is the URL for the full details of the new `MailingLIst`.
- `"name":"New List"`: This shows the name of the new list that we requested.
- `"owner":2`: This shows the ID of the owner of the list. This matches the ID of the user we created earlier and included in this request (using `-u`).
- `"subscriber_set":[]`: This shows that there are no subscribers in this mailing list.

We can now repeat our initial request to list `MailingLists` and check whether our new `MailingList` is included:

```
$ curl "http://localhost:8000/mailinglist/api/v1/mailing-list" \
     -u 'user:secret'
[{"url":"http://localhost:8000/mailinglist/api/v1/mailinglist/cd983e25-c6c8
-48fa-9afa-1fd5627de9f1","id":"cd983e25-
c6c8-48fa-9afa-1fd5627de9f1","name":"New
List","owner":2,"subscriber_set":[]}]
```

Seeing that we can run our server and API in development is great, but we don't want to always rely on manual testing. Let's take a look at how to automate testing our API next.

If you want to test creating subscribers, make sure that your Celery broker (for example, Redis) is running and that you've got a worker consuming tasks to get the full experience.

Testing your API

APIs provide value to our users by letting them automate their interactions with our service. Naturally, DRF helps us automate testing our code as well.

DRF provides replacements for all the common Django tools we discussed in `Chapter 8`, *Testing Answerly*:

- `APIRequestFactory` for Django's `RequestFactory` class
- `APIClient` for Django's `Client` class
- `APITestCase` for Django's `TestCase` class

`APIRequestFactory` and `APIClient` make it easier to send requests formatted for our API. For example, they provide an easy way to set credentials for a request that isn't relying on session-based authentication. Otherwise, the two classes serve the same purpose as their default Django equivalents.

The `APITestCase` class simply extends Django's `TestCase` class and replaces Django's `Client` with `APIClient`.

Let's take a look at an example that we can add to `django/mailinglist/tests.py`:

```
class ListMailingListsWithAPITestCase(APITestCase):

    def setUp(self):
        password = 'password'
        username = 'unit test'
        self.user = get_user_model().objects.create_user(
            username=username,
            password=password
        )
        cred_bytes = '{}:{}'.format(username, password).encode('utf-8')
        self.basic_auth = base64.b64encode(cred_bytes).decode('utf-8')
```

```
    def test_listing_all_my_mailing_lists(self):
        mailing_lists = [
            MailingList.objects.create(
                name='unit test {}'.format(i),
                owner=self.user)
            for i in range(3)
        ]

        self.client.credentials(
            HTTP_AUTHORIZATION='Basic {}'.format(self.basic_auth))

        response = self.client.get('/mailinglist/api/v1/mailing-list')

        self.assertEqual(200, response.status_code)
        parsed = json.loads(response.content)
        self.assertEqual(3, len(parsed))

        content = str(response.content)
        for ml in mailing_lists:
            self.assertIn(str(ml.id), content)
            self.assertIn(ml.name, content)
```

Let's take a closer look at the new code introduced in our `ListMailingListsWithAPITestCase` class:

- `class ListMailingListsWithAPITestCase(APITestCase)`: This makes `APITestCase` our parent class. The `APITestCase` class is basically a `TestCase` class with an `APIClient` object instead of the regular Django `Client` object assigned to the `client` attribute. We will use this class to test our view.
- `base64.b64encode(...)`: This does a base64 encoding of our username and password. We'll use this to provide an HTTP basic authentication header. We must use `base64.b64encode()` instead of `base64.base64()` because the latter introduces white space to visually break up long strings. Also, we will need to `encode`/`decode` our strings because `b64encode()` operates on `byte` objects.
- `client.credentials()`: This lets us set an authentication header to be sent all future requests by this `client` object. In our case, we're sending an HTTP basic authentication header.
- `json.loads(response.content)`: This parses the content body of the response return a Python list.
- `self.assertEqual(3, len(parsed))`: This confirms that the number of items in the parsed list is correct.

If we were to send a second request using `self.client`, we would not need to re-authenticate because `client.credentials()` remembers what it received and continues passing it to all requests. We can clear the credentials by calling `client.credentials()`.

Now, we know how to test our API code!

Summary

In this chapter, we covered how to use the Django REST framework to create an RESTful API for our Django project. We saw how the Django REST framework uses similar principles to Django forms and Django generic views. We also used some of the core classes in the Django REST framework, we used a `ModelSerializer` to build a `Serializer` based on a Django models, and we used a `ListCreateAPIView` to create a view that can list and create Django models We used `RetrieveUpdateDestroyAPIView` to manage a Django model instance based on its primary key.

Next, we'll deploy our code to the internet using Amazon Web Services.

13
Deploying Mail Ape

In this chapter, we will deploy Mail Ape onto a virtual machine in the **Amazon Web Services** (**AWS**) cloud. AWS is composed of many different services. We've already discussed using S3 and launching a container into AWS. In this chapter, we will use a lot more AWS services. We will use the **Relational Database Service (RDS)** for a PostgreSQL database server. We will use the **Simple Queue Service (SQS)** for our Celery message queue. We will use **Elastic Computing Cloud (EC2)** to run virtual machines in the cloud. Finally, we will use CloudFormation to define our infrastructure as code.

In this chapter, we will do the following things:

- Separate production and development settings
- Use Packer to create an Amazon Machine Image of our release
- Use CloudFormation to define the infrastructure as code
- Launch Mail Ape into AWS using the command line

Let's start by separating our production development settings.

Separating development and production

So far, we've kept a single requirements file and a single `settings.py` file. This has made development convenient. However, we can't use our development settings in production.

The current best practice is to have a separate file per environment. Each environment's file then imports a common file with shared values. We'll use this pattern for our requirements and settings files.

Let's start by splitting up our requirements files.

Separating our requirements files

To separate our requirements, we'll delete the existing `requirements.txt` file and replace it with common, development, and production requirements files. After we delete `requirements.txt`, let's create `requirements.common.txt` at the root of our project:

```
django<2.1
psycopg2<2.8
django-markdownify==0.3.0
django-crispy-forms==1.7.0
celery<4.2
django-celery-results<2.0
djangorestframework<3.8
factory_boy<3.0
```

Next, let's create a requirements file for `requirements.development.txt`:

```
-r requirements.common.txt
celery[redis]
```

Since we only use Redis in our development setup, we'll keep the package in our development requirements file.

We'll put our production requirements in `requirements.production.txt` at the root of the project:

```
-r requirements.common.txt
celery[sqs]
boto3
pycurl
```

In order for Celery to work with SQS (the AWS message queue service), we will need to install the Celery SQS library (`celery[sqs]`). We will also install `boto3`, the Python AWS library, and `pycurl`, a Python `curl` implementation.

Next, let's separate our Django settings files.

Creating common, development, and production settings

As in our previous chapters, before we sort our settings into three files, we'll create `common_settings.py` by renaming our current `settings.py` then making some changes.

Let's change `DEBUG = False` so that no new settings file can *accidentally* be in debug mode. Then, let's change the secret key to be obtained from an environment variable by updating `SECRET_KEY = os.getenv('DJANGO_SECRET_KEY').`

In the database config, we can remove all the credentials but keep the `ENGINE` (to make it clear that we intend to use Postgres everywhere):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
    }
}
```

Next, let's create a development settings file in `django/config/development_settings.py`:

```
from .common_settings import *

DEBUG = True

SECRET_KEY = 'secret key'

DATABASES['default']['NAME'] = 'mailape'
DATABASES['default']['USER'] = 'mailape'
DATABASES['default']['PASSWORD'] = 'development'
DATABASES['default']['HOST'] = 'localhost'
DATABASES['default']['PORT'] = '5432'

MAILING_LIST_FROM_EMAIL = 'mailape@example.com'
MAILING_LIST_LINK_DOMAIN = 'http://localhost'

EMAIL_HOST = 'smtp.example.com'
EMAIL_HOST_USER = 'username'
EMAIL_HOST_PASSWORD = os.getenv('EMAIL_PASSWORD')
EMAIL_PORT = 587
EMAIL_USE_TLS = True

CELERY_BROKER_URL = 'redis://localhost:6379/0'
```

Remember that you need to change your `MAILING_LIST_FROM_EMAIL`, `EMAIL_HOST` and `EMAIL_HOST_USER` to your correct development values.

Next, let's put our production settings in `django/config/production_settings.py`:

```
from .common_settings import *

DEBUG = False

assert SECRET_KEY is not None, (
    'Please provide DJANGO_SECRET_KEY environment variable with a value')

ALLOWED_HOSTS += [
    os.getenv('DJANGO_ALLOWED_HOSTS'),
]

DATABASES['default'].update({
    'NAME': os.getenv('DJANGO_DB_NAME'),
    'USER': os.getenv('DJANGO_DB_USER'),
    'PASSWORD': os.getenv('DJANGO_DB_PASSWORD'),
    'HOST': os.getenv('DJANGO_DB_HOST'),
    'PORT': os.getenv('DJANGO_DB_PORT'),
})

LOGGING['handlers']['main'] = {
    'class': 'logging.handlers.WatchedFileHandler',
    'level': 'DEBUG',
    'filename': os.getenv('DJANGO_LOG_FILE')
}

MAILING_LIST_FROM_EMAIL = os.getenv('MAIL_APE_FROM_EMAIL')
MAILING_LIST_LINK_DOMAIN = os.getenv('DJANGO_ALLOWED_HOSTS')

EMAIL_HOST = os.getenv('EMAIL_HOST')
EMAIL_HOST_USER = os.getenv('EMAIL_HOST_USER')
EMAIL_HOST_PASSWORD = os.getenv('EMAIL_HOST_PASSWORD')
EMAIL_PORT = os.getenv('EMAIL_HOST_PORT')
EMAIL_USE_TLS = os.getenv('EMAIL_HOST_TLS', 'false').lower() == 'true'

CELERY_BROKER_TRANSPORT_OPTIONS = {
    'region': 'us-west-2',
    'queue_name_prefix': 'mailape-',
CELERY_BROKER_URL = 'sqs://'
}
```

Our production settings file gets most of its values from environment variables so that we don't check production values into our server. There are three settings we need to review, as follows:

- `MAILING_LIST_LINK_DOMAIN`: This is the domain that links in our emails will point to. In our case, in the preceding code snippet, we used the same domain that we added to our `ALLOWED_HOSTS` list, ensuring that we're serving the domain that the links point to.
- `CELERY_BROKER_TRANSPORT_OPTIONS`: This is a dictionary of options that configure Celery to use the correct SQS queue. We will need to set the region to `us-west-2` because our entire production deployment will be in that region. By default, Celery will want to use a queue called `celery`. However, we don't want that name to collide with other Celery projects we might deploy. To prevent name collisions, we will configure Celery to use the `mailape-` prefix.
- `CELERY_BROKER_URL`: This tells Celery which broker to use. In our case, we're using SQS. We will give our virtual machine access to SQS using AWS's role-based authorization so that we don't have to provide any credentials.

Now that we have our production settings created, let's make our infrastructure in the AWS Cloud.

Creating an infrastructure stack in AWS

In order to host an app on AWS, we will need to ensure that we have some infrastructure set up. We'll need to the following things:

- A PostgreSQL server
- Security Groups to open network ports so that we can access our database and web server
- An InstanceProfile to give our deployed VM access to SQS

We could create all that using the AWS web console or using the command-line interface. However, over time, it can be hard to track how our infrastructure is configured if we rely on runtime tweaks. It would be much nicer if we could describe our required infrastructure in files that we could track in version control, much like we track our code.

AWS provides a service called CloudFormation, which lets us treat infrastructure as code. We will define our infrastructure in a CloudFormation template using YAML (JSON is also available, but we'll use YAML). We'll then execute our CloudFormation template to create a CloudFormation stack. The CloudFormation stack will be associated with actual resources in the AWS Cloud. If we delete the CloudFormation stack, the related resources will also be deleted. This gives us simple control over our use of AWS resources.

Let's create our CloudFormation template in `cloudformation/infrastructure.yaml`. Every CloudFormation template begins with a `Description` and the template format version information. Let's start our file with the following:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
```

Our CloudFormation template will have the following three parts:

- `Parameters`: This is where we will describe values that we'll pass in at runtime. This block is optional but useful. In our case, we'll pass in the master database password rather than hardcoding it in our template.
- `Resources`: This is where we will describe the specific resources that our stack will contain. This will describe our database server, SQS queue, security groups, and InstanceProfile.
- `Outputs`: This is where we will describe the values to output to make referencing the resources we created easier. This block is optional but useful. We will provide the address of our database server and the ID of the InstanceProfile we created.

Let's start by creating the `Parameters` block of our CloudFormation template.

Accepting parameters in a CloudFormation template

To avoid hardcoding values in a CloudFormation template, we can accept parameters. This helps us avoid hardcoding sensitive values (such as passwords) in a template.

Let's add a parameter to accept the password of our database server's master user:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  MasterDBPassword:
    Description: Master Password for the RDS instance
    Type: String
```

This adds a `MasterDBPassword` parameter to our template. We will be able to reference this value later on. CloudFormation templates let us add two pieces of information for parameters:

- `Description`: This is not used by CloudFormation but is useful for the people who have to maintain our infrastructure.
- `Type`: CloudFormation uses this to check whether the value we provide is valid *before* executing our template. In our case, the password is a `String`.

Next let's add a `Resources` block to define the AWS resources we'll need in our infrastructure.

Listing resources in our infrastructure

Next, we will add a `Resources` block to our CloudFormation template in `cloudformation/infrastructure.yaml`. Our infrastructure template will define five resources:

- Security Groups, which will open network ports, permitting us to access our database and web servers
- Our database server
- Our SQS queue
- A Role that allows access to SQS
- An InstanceProfile, which lets our web servers assume the above Role

Let's start by creating the Security Groups, which will open the network ports by which we'll access our database and web servers.

Adding Security Groups

In AWS, a SecurityGroup defines a set of network access rules much like a network firewall. By default, virtual machines launched can *send* data out on any network port but not *accept* connections on any network port. That means that we can't connect using SSH or HTTP; let's fix that.

Let's update our CloudFormation template in `cloudformation/infrastructure.yaml` with three new Security Groups:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  ...
Resources:
  SSHSecurityGroup:
    Type: "AWS::EC2::SecurityGroup"
    Properties:
      GroupName: ssh-access
      GroupDescription: permit ssh access
      SecurityGroupIngress:
        -
          IpProtocol: "tcp"
          FromPort: "22"
          ToPort: "22"
          CidrIp: "0.0.0.0/0"
  WebSecurityGroup:
    Type: "AWS::EC2::SecurityGroup"
    Properties:
      GroupName: web-access
      GroupDescription: permit http access
      SecurityGroupIngress:
        -
          IpProtocol: "tcp"
          FromPort: "80"
          ToPort: "80"
          CidrIp: "0.0.0.0/0"
  DatabaseSecurityGroup:
    Type: "AWS::EC2::SecurityGroup"
    Properties:
      GroupName: db-access
      GroupDescription: permit db access
```

```
SecurityGroupIngress:
  -
    IpProtocol: "tcp"
    FromPort: "5432"
    ToPort: "5432"
    CidrIp: "0.0.0.0/0"
```

In the preceding code block, we defined three new SecurityGroups to open ports `22` (SSH), `80` (HTTP), and `5432` (default Postgres port).

Let's take a closer look at the syntax of a CloudFormation resource. Each resource block must have a `Type` and `Properties` attributes. The `Type` attribute tells CloudFormation what this resource describes. The `Properties` attribute describe settings for this particular resources.

Our SecurityGroups used the following properties:

- `GroupName`: This gives human-friendly names. This is optional but recommended. CloudFormation can generate the names for us instead. SecurityGroup names must be unique for a given account (for example, I can't have two `db-access` groups, but you and I can each have a `db-access` group).
- `GroupDescription`: This is a human-friendly description of the group's purpose. It's required to be present.
- `SecurityGroupIngress`: This is a list of ports on which to accept incoming connection for VMs in this group.
- `FromPort`/`ToPort`: Often, these two settings will have the same value, the network port you want to be to be able to connect to. `FromPort` is the port on which we will connect. `ToPort` is the VM port on which a service is listening.
- `CidrIp`: This is an IPv4 range to accept connections from. `0.0.0.0/0` means accept all connections.

Next, let's add a database server to our list of resources.

Adding a Database Server

AWS offers relational databases servers as a service called **Relational Database Service** (**RDS**). To create a database server on AWS, we will create a new RDS VM (called an *instance*). One important thing to note is that when we launch an RDS instance, we can connect to the PostgreSQL database on the server but we do not have shell access. We must run Django on a different VM.

Let's add an RDS instance to our CloudFormation template in `cloudformation/infrastructure.yaml`:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  ...
Resources:
  ...
  DatabaseServer:
    Type: AWS::RDS::DBInstance
    Properties:
      DBName: mailape
      DBInstanceClass: db.t2.micro
      MasterUsername: master
      MasterUserPassword: !Ref MasterDBPassword
      Engine: postgres
      AllocatedStorage: 20
      PubliclyAccessible: true
      VPCSecurityGroups: !GetAtt DatabaseSecurityGroup.GroupId
```

Our new RDS instance entry is of the `AWS::RDS::DBInstance` type. Let's review the properties we set:

- `DBName`: This is the name of the *server*, not the name of any databases running on it.
- `DBInstanceClass`: This defines the memory and processing power of the virtual machine of the server. At the time of writing this book, `db.t2.micro` is part of a free tier for accounts in their first year.
- `MasterUsername`: This is the username of the privileged administrator account on the server.
- `MasterUserPassword`: This is the password for the privileged administrator account
- `!Ref MasterDBPassword`: This is the shortcut syntax to reference the `MasterDBPassword` parameter. This lets us avoid hardcoding the database server's administrator password.
- `Engine`: This is the type of Database server we want; in our case, `postgres` will give us a PostgreSQL server.
- `AllocatedStorage`: This indicates how much storage space the server should have, in gigabyte (GB).
- `PubliclyAccessible`: This indicates whether the server can be accessed from outside the AWS Cloud.

- `VPCSecurityGroups`: This is a list of SecurityGroups, indicating which ports are open and accessible.
- `!GetAtt DatabaseSecurityGroup.GroupId`: This returns the `GroupID` attribute of the `DatabaseSecurityGroup` security group.

This block also introduces us to CloudFormation's `Ref` and `GetAtt` functions. Both these functions let us reference other parts of our CloudFormation stack which is very important. `Ref` lets us use our `MasterDBPassword` parameter as the value of our database server's `MasterUserPassword`. `GetAtt` lets us reference our AWS generated `GroupId` attribute of `DatabaseSecurityGroup` in our database server's list of `VPCSercurityGroups`.

AWS CloudFormation offers a variety of different functions to make building templates easier. They are documented in the AWS documentation online (`https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference.html`).

Next, let's create the SQS Queue that Celery will use.

Adding a Queue for Celery

SQS is the AWS message queue service. Using SQS, we can create a Celery-compatible message queue that we don't have to maintain. SQS can quickly scale to handle any number of requests we send it.

To define our queue, add it to our `Resources` block in `cloudformation/infrastructure.yaml`:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  ...
Resources:
  ...
  MailApeQueue:
    Type: "AWS::SQS::Queue"
    Properties:
      QueueName: mailape-celery
```

Our new resource is of the `AWS::SQS::Queue` type and has a single property, `QueueName`.

Next, let's create a role and InstanceProfile to let our production servers access our SQS queue.

Creating a Role for Queue access

Earlier, in the *Adding Security Groups* section, we discussed creating SecurityGroups to open network ports so that we could make a network connection. To manage access among AWS resources, we will need to use role-based authorization. In a role-based authorization, we define a role, who can be assigned that role (assume that role), and what actions that role can perform. In order for our web servers to use that role, we will need to create an EC2 instance profile that is associated with that role.

Let's start by adding a role to `cloudformation/infrastructure.yaml`:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  ...
Resources:
  ...
   SQSAccessRole:
    Type: "AWS::IAM::Role"
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          -
            Effect: "Allow"
            Principal:
              Service:
                - "ec2.amazonaws.com"
            Action:
              - "sts:AssumeRole"
      Policies:
        -
          PolicyName: "root"
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              -
                Effect: Allow
                Action: "sqs:*"
                Resource: !GetAtt MailApeQueue.Arn
```

```
      -
        Effect: Allow
        Action: sqs:ListQueues
        Resource: "*"
```

Our new block is of the `AWS::IAM::Role` type. IAM is short for AWS Identity and Access Management services. Our role is composed of the following two properties:

- `AssumeRolePolicyDocument`: This defines who may be assigned this role. In our case, we're saying that this role may be assumed by any object in Amazon's EC2 service. Later, we'll use it in our EC2 instances.
- `Policies`: This is a list of allowed (or denied) actions for this role. In our case, we're permitting all SQS operations (`sqs:*`) on our previously defined SQS Queue. We reference our queue by getting its `Arn`, **Amazon Resource Name** (**ARN**), with the `GetAtt` function. ARNs are Amazon's way of providing each resource on the Amazon cloud with a globally unique ID.

Now that we have our role, we can associate it with an `InstanceProfile` resource, which can be associated with our web servers:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  ...
Resources:
  ...
  SQSClientInstance:
    Type: "AWS::IAM::InstanceProfile"
    Properties:
      Roles:
        - !Ref SQSAccessRole
```

Our new InstanceProfile is of the `AWS::IAM::InstanceProfile` type and needs a list of associated roles. In our case, we simply reference our previously created `SQSAccessRole` using the `Ref` function.

Now that we've created our infrastructure resources, let's output the address of our database and our ARN of `InstanceProfile` resource.

Outputting our resource information

CloudFormation templates can have an output block to make it easier to reference the created resources. In our case, we will output the address of our database server and the ARN of `InstanceProfile`.

Let's update our CloudFormation template in `cloudformation/infrastructure.yaml`:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape Infrastructure
Parameters:
  ...
Resources:
  ...
Outputs:
  DatabaseDNS:
    Description: Public DNS of RDS database
    Value: !GetAtt DatabaseServer.Endpoint.Address
  SQSClientProfile:
    Description: Instance Profile for EC2 instances that need SQS Access
    Value: !GetAtt SQSClientInstance.Arn
```

In the preceding code, we're using the `GetAtt` function to return the address of our `DatabaseServer` resource and the ARN of our `SQSClientInstance InstanceProfile` resource.

Executing our template to create our resources

Now that we've created our `CloudFormation` template, we can create a `CloudFormation` stack. When we tell AWS to create our `CloudFormation` stack, it will create all the related resources in our template.

To create our template, we will need the following two things:

- The AWS **command-line interface** (**CLI**)
- An AWS access key/secret key pair

We can install the AWS CLI using `pip`:

```
$ pip install awscli
```

To get (or create) your access key/secret key pair, you will need access to the Security Credential (`https://console.aws.amazon.com/iam/home?region=us-west-2#/security_credential`) section of your AWS Console.

Then we need to configure the AWS command line tool with our key and region. The `aws` command offers an interactive `configure` subcommand to do this. Let's run it on the command line:

```
$ aws configure
AWS Access Key ID [None]: <Your ACCESS key>
AWS Secret Access Key [None]: <Your secret key>
Default region name [None]: us-west-2
Default output format [None]: json
```

The `aws configure` command stores the values you entered in a `.aws` directory in your home directory.

With those setups, we can now create our stack:

```
$ aws cloudformation create-stack \
    --stack-name "infrastructure" \
    --template-body
"file:///path/to/mailape/cloudformation/infrastrucutre.yaml" \
    --capabilities CAPABILITY_NAMED_IAM \
    --parameters \
      "ParameterKey=MasterDBPassword,ParameterValue=password" \
    --region us-west-2
```

Creating a stack can take some time, so the command returns without waiting for success. Let's take a closer look at our `create-stack` command:

- `--stack-name`: This is the name of stack we're creating. Stack names must be unique per account.
- `--template-body`: This is either the template itself, or, in our case, a `file://` URL to the template file. Remember that `file://` URLs require the absolute path to the file.
- `--capabilities CAPABILITY_NAMED_IAM`: This is required for templates that create or affect **Identity and Access Management** (**IAM**) services. This prevents accidentally affecting access management services.
- `--parameters`: This lets us pass in values for a template's parameters. In our case, we set the master password for our database as `password`, which is not a safe value.

- `--region`: The AWS Cloud is organized as a collection of regions across the world. In our case, we're using `us-west-2`, which is located in a series of data centers around the US state of Oregon.

Remember that you need to set a secure master password for your database.

To take a look at how stack creation is doing, we can check it using AWS Web Console (`https://us-west-2.console.aws.amazon.com/cloudformation/home?region=us-west-2`) or using the command line:

```
$ aws cloudformation describe-stacks \
    --stack-name "infrastructure" \
    --region us-west-2
```

When the stack is done creating the related resources, it will return a result similar to this:

```
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-
west-2:XXX:stack/infrastructure/NNN",
            "StackName": "infrastructure",
            "Description": "Mail Ape Infrastructure",
            "Parameters": [
                {
                    "ParameterKey": "MasterDBPassword",
                    "ParameterValue": "password"
                }
            ],
            "StackStatus": "CREATE_COMPLETE",
            "Outputs": [
                {
                    "OutputKey": "SQSClientProfile",
                    "OutputValue": "arn:aws:iam::XXX:instance-
profile/infrastructure-SQSClientInstance-XXX",
                    "Description": "Instance Profile for EC2 instances that
need SQS Access"
                },
                {
                    "OutputKey": "DatabaseDNS",
                    "OutputValue": "XXX.XXX.us-west-2.rds.amazonaws.com",
                    "Description": "Public DNS of RDS database"
                }
```

```
                ],
            }
        ]
    }
```

Two things to pay particular attention to in the `describe-stack` result are as follows:

- The object under the `Parameters` key, which will show our master database password in plain text
- The object `Outputs` key, which shows the ARN of our `InstanceProfile` resource and the address of our database server

In all the previous code, I've replaced values specific to my account with XXX. Your output will differ.

If you want to remove the resources associated with your stack, you can just delete the stack:

```
$ aws cloudformation delete-stack --stack-name "infrastructure"
```

Next, we'll build an Amazon Machine Image, which we'll use to run Mail Ape in AWS.

Building an Amazon Machine Image with Packer

Now that we have our infrastructure running in AWS, let's build our Mail Ape server. In AWS, we could launch an official Ubuntu VM, follow the steps in `Chapter 9`, *Deploying Answerly*, and have our Mail Ape running. However, AWS treats EC2 instances as *ephemeral*. If an EC2 instance gets terminated, then we will have to launch a new instance and configure it all over again. There are a few ways to mitigate this problem. We'll solve the problem of ephemeral EC2 instances by building a new **Amazon Machine Image** (**AMI**) for our release. Then any time we launch an EC2 instance using that AMI, it will be already perfectly configured.

We will automate building our AMIs using a HashiCorp's Packer tool. Packer gives us a way of creating an AMI from a Packer template. A Packer template is a JSON file that defines the steps needed to configure our EC2 instance into our desired state and save the AMI. For our Packer template to run, we'll also write a collection of shell scripts to configure our AMI. Using a tool like Packer, we can automate building a new release AMI.

Let's start by installing Packer on our machines.

Installing Packer

Get Packer from the `https://www.packer.io` download page. Packer is available for all major platforms.

Next, we'll create a script to make the directories we'll rely on in production.

Creating a script to create our directory structure

The first script we will write will create directories for all our code. Let's add the following script to our project in `scripts/make_aws_directories.sh`:

```
#!/usr/bin/env bash
set -e

sudo mkdir -p \
    /mailape/ubuntu \
    /mailape/apache \
    /mailape/django \
    /var/log/celery \
    /etc/mailape \
    /var/log/mailape

sudo chown -R ubuntu /mailape
```

In the preceding code, we're using `mkdir` to make the directories. Next, we want to make the `ubuntu` user can write to the `/mailape` directories so that we recursively `chown` the `/mailape` directory.

So, let's create a script to install the Ubuntu packages we require.

Creating a script to install all our packages

In our production environment, we will have to install Ubuntu packages as well as the Python packages we've already listed. First, let's list all our Ubuntu packages in `ubuntu/packages.txt`:

```
python3
python3-pip
```

```
python3-dev
virtualenv
apache2
libapache2-mod-wsgi-py3
postgresql-client
libcurl4-openssl-dev
libssl-dev
```

Next, let's create a script to install all the packages in `scripts/install_all_packages`:

```
#!/usr/bin/env bash
set -e

sudo apt-get update
sudo apt install -y $(cat /mailape/ubuntu/packages.txt | grep -i '^[a-z]')

virtualenv -p $(which python3) /mailape/virtualenv
source /mailape/virtualenv/bin/activate

pip install -r /mailape/requirements.production.txt

sudo chown -R www-data /var/log/mailape \
    /etc/mailape \
    /var/run/celery \
    /var/log/celery
```

In the preceding script, we will install the Ubuntu packages that we listed above, then create a `virtualenv` to isolate our Mail Ape Python environment and packages. Finally, we give Apache (the `www-data` user) the ownership of some directories so that it can write to them. We couldn't give the `www-data` user the ownership because they probably didn't exist until we installed the `apache2` package.

Next, let's configure to run Apache2 to run Mail Ape using mod_wsgi.

Configuring Apache

Now, we'll add Apache mod_wsgi configuration just like we did in `Chapter 9`, *Deploying Answerly*. The mod_wsgi configuration isn't the focus of this chapter, so refer to `Chap`ter 9, *Deploying Answerly*, for details of how this configuration works.

Let's create a virtual host configuration file for Mail Ape in `apache/mailape.apache.conf`:

```
LogLevel info
WSGIRestrictEmbedded On

<VirtualHost *:80>

    WSGIDaemonProcess mailape \
        python-home=/mailape/virtualenv \
        python-path=/mailape/django \
        processes=2 \
        threads=2

    WSGIProcessGroup mailape

    WSGIScriptAlias / /mailape/django/config/wsgi.py
    <Directory /mailape/django/config>
        <Files wsgi.py>
            Require all granted
        </Files>
    </Directory>

    Alias /static/ /mailape/django/static_root
    <Directory /mailape/django/static_root>
        Require all granted
    </Directory>
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

</VirtualHost>
```

As we discussed in `Chapter 9`, *Deploying Answerly*, there isn't a way to pass environment variables to our mod_wsgi Python processes, so we will need to update our project's `wsgi.py` just like we did in `Chapter 9`, *Deploying Answerly*.

Here's our new `django/config/wsgi.py`:

```
import os
import configparser

from django.core.wsgi import get_wsgi_application

if not os.environ.get('DJANGO_SETTINGS_MODULE'):
    parser = configparser.ConfigParser()
    parser.read('/etc/mailape/mailape.ini')
    for name, val in parser['mod_wsgi'].items():
```

```
        os.environ[name.upper()] = val

application = get_wsgi_application()
```

We discussed the preceding script in Chapter 9, *Deploying Answerly*. The only difference here is the file we parse, that is, `/etc/mailape/mailape.ini`.

Next, we will need to add our virtual host configuration to the Apache `sites-enabled` directory. Let's create a script to do that in `scripts/configure_apache.sh`:

```
#!/usr/bin/env bash

sudo rm /etc/apache2/sites-enabled/*
sudo ln -s /mailape/apache/mailape.apache.conf /etc/apache2/sites-
enabled/000-mailape.conf
```

Now that we have a script to configure Apache in a production environment, let's configure our Celery workers to start.

Configuring Celery

Now that we have Apache running Mail Ape, we will need to configure Celery to start and process our SQS queue. To start our Celery workers, we will use Ubuntu's systemd process management facility.

First, let's create a Celery service file to tell SystemD how to start Celery. We'll create the service file in `ubuntu/celery.service`:

```
[Unit]
Description=Mail Ape Celery Service
After=network.target

[Service]
Type=forking
User=www-data
Group=www-data
EnvironmentFile=/etc/mailape/celery.env
WorkingDirectory=/mailape/django
ExecStart=/bin/sh -c '/mailape/virtualenv/bin/celery multi start worker \
    -A "config.celery:app" \
    --logfile=/var/log/celery/%n%I.log --loglevel="INFO" \
    --pidfile=/run/celery/%n.pid'
ExecStop=/bin/sh -c '/mailape/virtualenv/bin/celery multi stopwait worker \
    --pidfile=/run/celery/%n.pid'
ExecReload=/bin/sh -c '/mailape/virtualenv/bin/celery multi restart worker
```

```
\
   -A "config.celery:app" \
   --logfile=/var/log/celery/%n%I.log --loglevel="INFO" \
   --pidfile=/run/celery/%n.pid'

[Install]
WantedBy=multi-user.target
```

Let's take a closer look at some of the options in this file:

- `After=network.target`: This means that SystemD should not start this until our server has connected to a network.
- `Type=forking`: This means that the `ExecStart` command will eventually start a new process that continues to run under its own process ID (PID).
- `User`: This indicates the user that will own the Celery processes. In our case, we're just going to reuse Apache's `www-data` user.
- `EnvironmentFile`: This lists a file that will be read for environment variables and values that will be set for all the `Exec` commands. We list one with our Celery configuration (`/mailape/ubuntu/celery.systemd.conf`) and one with our Mail Ape configuration (`/etc/mailape/celery.env`).
- `ExecStart`: This is the command that will be executed to start Celery. In our case, we start multiple Celery workers. All our Celery commands will operate on our workers based on the process ID files they create. Celery will replace `%n` with the worker's ID.
- `ExecStop`: This is the command that will be executed to stop our Celery workers, based on their PID files.
- `ExecReload`: This is the command that will be executed to restart our Celery workers. Celery supports a `restart` command, so we will use that to perform the restart. However, this command must receive the same options as our `ExecStart` command.

We'll be placing our PID files in `/var/run/celery`, but we will need to make sure that the directory is created. `/var/run` is a special directory, which doesn't use a regular filesystem. We'll need to create a configuration file to tell Ubuntu to create `/var/run/celery`. Let's create this file in `ubuntu/tmpfiles-celery.conf`:

```
d    /run/celery   0755 www-data www-data - -
```

This tells Ubuntu to create a directory, `/run/celery`, owned by the Apache user (`www-data`).

Finally, let's create a script to put all these files in the right places on our server. We'll name this script `scripts/configure_celery.sh`:

```
#!/usr/bin/env bash

sudo ln -s /mailape/ubuntu/celery.service
/etc/systemd/system/celery.service
sudo ln -s /mailape/ubuntu/celery.service /etc/systemd/system/multi-
user.target.wants/celery.service
sudo ln -s /mailape/ubuntu/tmpfiles-celery.conf /etc/tmpfiles.d/celery.conf
```

Now that Celery and Apache are configured, let's make sure that they have the correct environment configuration to run Mail Ape

Creating the environment configuration files

Both our Celery and mod_wsgi Python processes will need to pull configuration information out of the environment to connect to the right database, SQS Queue, and many other services. These are settings and values we don't want to check into our version control system (for example, passwords). However, we still need them to be set in a production environment. To create the files that define the environment that our processes will run in, we'll make the script in `scripts/make_mailape_environment_ini.sh`:

```
#!/usr/bin/env bash

ENVIRONMENT="
DJANGO_ALLOWED_HOSTS=${WEB_DOMAIN}
DJANGO_DB_NAME=mailape
DJANGO_DB_USER=mailape
DJANGO_DB_PASSWORD=${DJANGO_DB_PASSWORD}
DJANGO_DB_HOST=${DJANGO_DB_HOST}
DJANGO_DB_PORT=5432
DJANGO_LOG_FILE=/var/log/mailape/mailape.log
DJANGO_SECRET_KEY=${DJANGO_SECRET}
DJANGO_SETTINGS_MODULE=config.production_settings
MAIL_APE_FROM_EMAIL=admin@blvdplatform.com
EMAIL_HOST=${EMAIL_HOST}
EMAIL_HOST_USER=mailape
EMAIL_HOST_PASSWORD=${EMAIL_HOST_PASSWORD}
EMAIL_HOST_PORT=587
EMAIL_HOST_TLS=true
```

```
INI_FILE="[mod_wsgi]
${ENVIRONMENT}
"

echo "${INI_FILE}" | sudo tee "/etc/mailape/mailape.ini"
echo "${ENVIRONMENT}" | sudo tee "/etc/mailape/celery.env"
```

Our `make_mailape_environment_ini.sh` script has some values hardcoded but references others (for example, passwords) as environment variables. We'll pass the values for these variables into Packer at runtime. Packer will then pass these values to our script.

Next, let's make our Packer template to build our AMI.

Making a Packer template

Packer creates an AMI based on the instructions listed in a Packer template file. A Packer template is a JSON file made up of three top-level keys:

- `variables`: This will let us set values (such as passwords) at runtime
- `builders`: This specifies the cloud platform-specific details, such as AWS credentials
- `provisioners`: This are the instructions Packer will execute to make our image

Let's create our Packer template in `packer/web_worker.json`, starting with the `variables` section:

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": "",
    "django_db_password":"",
    "django_db_host":"",
    "django_secret":"",
    "email_host":"",
    "email_host_password":"",
    "mail_ape_aws_key":"",
    "mail_ape_secret_key":"",
    "sqs_celery_queue":"",
    "web_domain":""
  }
}
```

Under the `variables` key, we will list all the variables we want our template to take as the keys of JSON object. If the variable has a default value, then we can provide it as the value for that variable's key.

Next, let's add a `builders section` to configure Packer to use AWS:

```
{
  "variables": {...},
  "builders": [
    {
      "type": "amazon-ebs",
      "access_key": "{{user `aws_access_key`}}",
      "secret_key": "{{user `aws_secret_key`}}",
      "region": "us-west-2",
      "source_ami": "ami-78b82400",
      "instance_type": "t2.micro",
      "ssh_username": "ubuntu",
      "ami_name": "mailape-{{timestamp}}",
      "tags": {
        "project": "mailape"
      }
    }
  ]
}
```

A `builders` is an array because we could use the same template to build a machine image on multiple platforms (for example, AWS and Google Cloud). Let's take a look at each option in detail:

- `"type": "amazon-ebs"`: Tells Packer we're creating an Amazon Machine Image with Elastic Block Storage. This is the preferred configuration due to the flexibility it offers.
- `"access_key": "{{user aws_access_key }}"`: This is the access key Packer should use to authenticate itself with AWS. Packer includes its own template language so that values can be generated at runtime. Any value between `{{ }}` is generated by the Packer template engine. The template engine offers a `user` function, which takes the name of the user-provided variable and returns its value. For example, `{{user aws_access_key }}` will be replaced by the value the user provided to `aws_access_key` when running Packer.
- `"secret_key": "{{user aws_secret_key }}"`: This is the same for the AWS Secret Key.
- `"region": "us-west-2"`: This specifies the AWS region. All our work will be done in `us-west-2`.

- `"source_ami": "ami-78b82400"`: This is the image that we're going to customize to make our image. In our case, we're using an official Ubuntu AMI. Ubuntu offers an EC2 AMI locator (`http://cloud-images.ubuntu.com/locator/ec2/`) to help find their office AMIs.
- `"instance_type": "t2.micro"`: This is a small inexpensive instance that, at the time of writing this book, falls under the AWS free tier.
- `"ssh_username": "ubuntu"`: Packer performs all its operations on the virtual machine over SSH. This is the username it should use for authentication. Packer will generate its own key pair for authentication, so we don't have to worry about specifying a password or key.
- `"ami_name": "mailape-{{timestamp}}"`: The name of the resulting AMI. `{{timestamp}}` is a function that returns the UTC time in seconds since the Unix epoch.
- `"tags": {...}`: Tagging resources makes it easier to identify resources in AWS. This is optional but recommended.

Now that we've specified our AWS builder, we will need to specify our provisioners.

Packer provisioners are the instructions that customize the server. In our case, we will use the following two types of provisioners:

- `file` provisioners to upload our code to the server
- `shell` provisioners to execute our scripts and commands

First, let's add our `make_aws_directories.sh` script, as we'll need it to run first:

```
{
  "variables": {...},
  "builders": [...],
  "provisioners": [
    {
      "type": "shell",
      "script": "{{template_dir}}/../scripts/make_aws_directories.sh"
    }
  ]
}
```

A `shell` provisioner with a `script` property will upload, execute, and remove the script. Packer provides the `{{template_dir}}` function, which returns the directory of template directory. This lets us avoid hardcoding absolute paths. The first provisioner we execute will execute the `make_aws_directories.sh` script we created earlier in this section.

Now that our directories exist, let's copy our code and files over using `file` provisioners:

```
{
  "variables": {...},
  "builders": [...],
  "provisioners": [
    ...,
    {
      "type": "file",
      "source": "{{template_dir}}/../requirements.common.txt",
      "destination": "/mailape/requirements.common.txt"
    },
    {
      "type": "file",
      "source": "{{template_dir}}/../requirements.production.txt",
      "destination": "/mailape/requirements.production.txt"
    },
    {
      "type": "file",
      "source": "{{template_dir}}/../ubuntu",
      "destination": "/mailape/ubuntu"
    },
    {
      "type": "file",
      "source": "{{template_dir}}/../apache",
      "destination": "/mailape/apache"
    },
    {
      "type": "file",
      "source": "{{template_dir}}/../django",
      "destination": "/mailape/django"
    },
  ]
}
```

`file` provisioners upload local files or directories defined by `source` to the server at `destination`.

Since we uploaded our Python code from our working directory, we need to be careful of old `.pyc` files hanging around. Let's make sure that we delete those on our production server:

```
{
  "variables": {...},
  "builders": [...],
  "provisioners": [
    ...,
```

```
    {
       "type": "shell",
       "inline": "find /mailape/django -name '*.pyc' -delete"
    },
    ]
}
```

A `shell` provisioner can receive an `inline` attribute. The provisioner will then execute the `inline` command on the server.

Finally, let's execute the rest of the scripts we created:

```
{
  "variables": {...},
  "builders": [...],
  "provisioners": [
    ...,
    {
      "type": "shell",
      "scripts": [
        "{{template_dir}}/../scripts/install_all_packages.sh",
        "{{template_dir}}/../scripts/configure_apache.sh",
        "{{template_dir}}/../scripts/make_mailape_environment_ini.sh",
        "{{template_dir}}/../scripts/configure_celery.sh"
        ],
      "environment_vars": [
        "DJANGO_DB_HOST={{user `django_db_host`}}",
        "DJANGO_DB_PASSWORD={{user `django_db_password`}}",
        "DJANGO_SECRET={{user `django_secret`}}",
        "EMAIL_HOST={{user `email_host`}}",
        "EMAIL_HOST_PASSWORD={{user `email_host_password`}}",
        "WEB_DOMAIN={{user `web_domain`}}"
      ]
}
```

In this case, the `shell` provisioner has received `scripts` and `environment_vars`. `scripts` is an array of paths to shell scripts. Each item in the array will be uploaded and executed. When executing each script, this `shell` provisioner will add the environment variables listed in `environment_vars`. The `environment_vars` parameter is optionally available to all `shell` provisioners to provide extra environment variables.

With our final provisioner added to our file, we've now finished our Packer template. Let's use Packer to execute the template and build our Mail Ape production server.

Running Packer to build an Amazon Machine Image

With Packer installed and a Mail Ape production server Packer template created, we're ready to build our **Amazon Machine Image** (**AMI**).

Let's run Packer to build our AMI:

```
$ packer build \
    -var "aws_access_key=..." \
    -var "aws_secret_key=..." \
    -var "django_db_password=..." \
    -var "django_db_host=A.B.us-west-2.rds.amazonaws.com" \
    -var "django_secret=..." \
    -var "email_host=smtp.example.com" \
    -var "email_host_password=..." \
    -var "web_domain=mailape.example.com" \
    packer/web_worker.json
Build 'amazon-ebs' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:
us-west-2: ami-XXXXXXXX
```

Packer will output the AMI ID of our new AMI image. We'll be able to use this AMI to launch an EC2 instance in the AWS Cloud.

If your template fails due to a missing Ubuntu package, retry the build. At the time of writing this book, the Ubuntu package repositories do not always update successfully.

Now that we have our AMI, we can deploy it.

Deploying a scalable self-healing web app on AWS

Now that we have our infrastructure and a deployable AMI, we can deploy Mail Ape on AWS. Rather than launching a single EC2 instance from our AMI, we will deploy our app using CloudFormation. We'll define a set of resources that will let us scale our app up and down as needed. We'll define the following three resources:

- An Elastic Load Balancer to distribute requests among our EC2 instances
- An AutoScaling Group to launch and terminate EC2 instances
- A LaunchConfig to describe what kind of EC2 instances to launch

First, let's make sure that we have an SSH key if we need to access any of our EC2 instances to troubleshoot any problems after we deploy.

Creating an SSH key pair

To create an SSH key pair in AWS, we can use the following AWS command line:

```
$ aws ec2 create-key-pair --key-name mail_ape_production --region us-west-2
{
    "KeyFingerprint": "XXX",
    "KeyMaterial": "-----BEGIN RSA PRIVATE KEY-----\nXXX\n-----END RSA
PRIVATE KEY-----",
    "KeyName": "tom-cli-test"
}
```

Ensure that you copy the `KeyMaterial` value to your SSH client's configuration directory (typically, `~/.ssh`)—remember to replace `\n` with actual new lines.

Next, let's start our Mail Ape deployment CloudFormation template.

Creating the web servers CloudFormation template

Next, let's create a CloudFormation template to deploy Mail Ape servers to the cloud. We'll use CloudFormation to tell AWS how to scale our servers and relaunch them, should a disaster strike. We'll tell CloudFormation to create the following three resources:

- An **Elastic Load Balancer** (**ELB**), which will be able to distribute requests among our servers
- A LaunchConfig, which will describe the AMI, instance type, and other details of the EC2 instances we want to use.
- An AutoScaling Group, which will monitor to ensure that we have the right number of healthy EC2 instances.

These three resources are the core of building any kind of scaling self-healing AWS application.

Let's start building our CloudFormation template in `cloudformation/web_worker.yaml`. Our new template will have the same three sections as `cloudformation/infrastracture.yaml`: `Parameters`, `Resources`, and `Outputs`.

Lets's start by adding the `Parameters` section.

Accepting parameters in the web worker CloudFormation template

Our web worker CloudFormation template will accept the AMI to launch and the InstanceProfile to be used as parameters. This means that we won't have to hardcode the names of the resources we created with Packer and our infrastructure stack, respectively.

Let's create our template in `cloudformation/web_worker.yaml`:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape web worker
Parameters:
  WorkerAMI:
    Description: Worker AMI
    Type: String
  InstanceProfile:
    Description: the instance profile
    Type: String
```

Now that we have the AMI and the InstanceProfile for our EC2 instances, let's create our CloudFormation stack's resources.

Creating Resources in our web worker CloudFormation template

Next, we'll definite the **Elastic Load Balancer** (**ELB**), Launch Config, and AutoScaling Group. These three resources are the core of most scalable AWS web applications. We'll take a look at how they interact as we build up our template.

First, let's add our Load Balancer:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape web worker
Parameters:
  ...
Resources:
  LoadBalancer:
    Type: "AWS::ElasticLoadBalancing::LoadBalancer"
    Properties:
      LoadBalancerName: MailApeLB
      Listeners:
        -
          InstancePort: 80
          LoadBalancerPort: 80
          Protocol: HTTP
```

In the preceding code, we're adding a new resource called `LoadBalancer` of the `AWS::ElasticLoadBalancing::LoadBalancer` type. An ELB needs a name (`MailApeLB`) and a list of `Listeners`. Each `Listeners` entry should define the port our ELB is listening on (`LoadBalancerPort`, the instance port the request will be forwarded to (`InstancePort`), and the protocol the port will use (in our case, `HTTP`).

An ELB will be responsible for distributing HTTP requests across however many EC2 instances we launch to handle our load.

Next, we'll create a LaunchConfig to tell AWS how to launch a new Mail Ape web worker:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape web worker
Parameters:
  ...
Resources:
  LoadBalancer:
```

```
  ...
LaunchConfig:
  Type: "AWS::AutoScaling::LaunchConfiguration"
  Properties:
    ImageId: !Ref WorkerAMI
    KeyName: mail_ape_production
    SecurityGroups:
      - ssh-access
      - web-access
    InstanceType: t2.micro
    IamInstanceProfile: !Ref InstanceProfile
```

A Launch Config is of the `AWS::AutoScaling::LaunchConfiguration` type and describes the configuration of a new EC2 instance that an Auto Scaling Group should launch. Let's go through all the `Properties` to ensure that we understand what they mean:

- `ImageId`: This is the ID of the AMI we want the instances to run. In our case, we're using the `Ref` function to get the AMI ID from the `WorkerAMI` parameter.
- `KeyName`: This is the name of the SSH key that will be added to this machine. This is useful if we ever need to troubleshoot something live. In our case, we're using the name of the SSH key pair we created earlier in this chapter.
- `SecurityGroups`: This is a list of Security Group names that define what ports AWS is to open. In our case, we're listing the names of the web and SSH groups we created in our infrastructure stack.
- `InstanceType`: This indicates the instance type of our EC2 instances. An instance type defines the computing and memory resources available to our EC2 instance. In our case, we're using a very small affordable instance that is (at the time of writing this book) covered by the AWS Free Tier during the first year.
- `IamInstanceProfile`: This indicates the `InstanceProfile` for our EC2 instances. Here, we're using the `Ref` function to reference the `InstanceProfile` parameter. When we create our stack, we'll use the ARN of the InstanceProfile we created earlier that gives our EC2 instances access to SQS.

Next, we'll define the AutoScaling Group that launches the EC2 instances that the Launch Config describes to serve requests forwarded by the ELB:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape web worker
Parameters:
  ...
Resources:
  LoadBalancer:
    ...
```

```
LaunchConfig:
  ...
WorkerGroup:
  Type: "AWS::AutoScaling::AutoScalingGroup"
  Properties:
    LaunchConfigurationName: !Ref LaunchConfig
    MinSize: 1
    MaxSize: 3
    DesiredCapacity: 1
    LoadBalancerNames:
      - !Ref LoadBalancer
```

Our new **Auto Scaling Group** (**ASG**) is of the `AWS::AutoScaling::AutoScalingGroup` type. Let's go through its properties:

- `LaunchConfigurationName`: This is the name of the `LaunchConfiguration` this ASG should use when launching new instances. In our case, we use the `Ref` function to reference `LaunchConfig`, the Launch Configuration we created above.
- `MinSize`/`MaxSize`: These are the attributes required that set the maximum and minimum number of instances this group may contain. These values protect us from accidentally deploying too many instances that may negatively affect either our system or our monthly bill. In our case, we make sure that there is at least one (`1`) instance but no more than three (`3`).
- `DesiredCapacity`: This tells our system how many ASG and how many healthy EC2 instances should be running this ASG. If an instance fails and brings the number of healthy instances below the `DesiredCapacity` value, then ASG will use its Launch Configuration to launch more instances.
- `LoadBalancerNames`: This is a list of ELBs that can route requests to the instances launched by this ASG. When a new EC2 instance becomes a part of this ASG, it will also be added to the list of instances the named ELBs route requests to. In our case, we use the `Ref` function to reference the ELB we defined earlier in this template.

These three tools work together to help us make our Django app scale quickly and smoothly. The ASG gives us a way of saying how many Mail Ape EC2 instances we want running. The Launch Config describes how to launch a new Mail Ape EC2 instance. The ELB will then distribute the requests to all the instances that the ASG launched.

Now that we have our resources, let's output some of the most relevant data to make the rest of our deployment easy.

Outputting resource names

The final section we'll add to our CloudFormation template is `Outputs` to make it easier to note the address of our ELB and the name of our ASG. We'll need the address of our ELB to add a CNAME record to `mailape.example.com`. We'll need the name of our ASG if we need to access our instances (for example, to run our migrations).

Let's update `cloudformation/web_worker.yaml` with an `Outputs` section:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape web worker
Parameters:
  ...
Resources:
  LoadBalancer:
    ...
  LaunchConfig:
    ...
  WorkerGroup:
    ...
Outputs:
  LoadBalancerDNS:
    Description: Load Balancer DNS name
    Value: !GetAtt LoadBalancer.DNSName
  AutoScalingGroupName:
    Description: Auto Scaling Group name
    Value: !Ref WorkerGroup
```

The value of `LoadBalancerDNS` will be the DNS name of the ELB we created above. The value of `AutoScalingGroupName` will be our ASG, which returns the name of the ASG.

Next, let's create a stack for our Mail Ape 1.0 release.

Creating the Mail Ape 1.0 release stack

Now that we have our Mail Ape web worker CloudFormation template, we can create a CloudFormation stack. When we create the stack, the stack will create its related resources such as the ELB, ASG, and Launch Config. We'll use the AWS CLI to create our stack:

```
$ aws cloudformation create-stack \
    --stack-name "mail_ape_1_0" \
    --template-body
"file:///path/to/mailape/cloudformation/web_worker.yaml" \
    --parameters \
      "ParameterKey=WorkerAMI,ParameterValue=AMI-XXX" \
"ParameterKey=InstanceProfile,ParameterValue=arn:aws:iam::XXX:instance-
profile/XXX" \
    --region us-west-2
```

The preceding command looks very similar to the one we executed to create our infrastructure stack, but there are a couple of differences:

- `--stack-name`: This is the name of the stack we're creating.
- `--template-body "file:///path/..."`: This is a `file://` URL with an absolute path to our CloudFormation template. Since the path prefix ends with two `/` and a Unix path starts with a `/`, we get a weird looking triple `/` here.
- `--parameters`: This template takes two parameters. We can provide them in any order, but we must provide both.
- `"ParameterKey=WorkerAMI, ParameterValue=`: For `WorkerAMI`, we must provide the AMI ID that Packer gave us.
- `"ParameterKey=InstanceProfile,ParameterValue`: For InstanceProfile, we must provide the Instance Profile ARN that our infrastructure stack output.
- `--region us-west-2`: We're doing all our work in the `us-west-2` region.

To take a look at our stack's outputs, we can use the `describe-stack` command from the AWS CLI:

```
$ aws cloudformation describe-stacks \
    --stack-name mail_ape_1_0 \
    --region us-west-2
```

The result is a large JSON object; here is a slightly truncated example version:

```
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-
west-2:XXXX:stack/mail_ape_1_0/XXX",
            "StackName": "mail_ape_1_0",
            "Description": "Mail Ape web worker",
            "Parameters": [
                {
                    "ParameterKey": "InstanceProfile",
                    "ParameterValue": "arn:aws:iam::XXX:instance-
profile/XXX"
                },
                {
                    "ParameterKey": "WorkerAMI",
                    "ParameterValue": "ami-XXX"
                }
            ],
            "StackStatus": "CREATE_COMPLETE",
            "Outputs": [
                {
                    "OutputKey": "AutoScalingGroupName",
                    "OutputValue": "mail_ape_1_0-WebServerGroup-XXX",
                    "Description": "Auto Scaling Group name"
                },
                {
                    "OutputKey": "LoadBalancerDNS",
                    "OutputValue": "MailApeLB-XXX.us-
west-2.elb.amazonaws.com",
                    "Description": "Load Balancer DNS name"
                }
            ],
        }
    ]
}
```

Our resources (for example, EC2 instances) won't be ready until `StackStatus` is `CREATE_COMPLETE`. It can take a few minutes to create all the relevant resources.

We're particularly interested in the objects in the `Outputs` array:

- The first value gives us the name of our ASG. With the name of our ASG, we'll be able to find the EC2 instances in that ASG in case we need to SSH into one.
- The second value gives us the DNS name of our ELB. We'll use our ELB's DNS to create CNAME record for our production DNS record so that we redirect our traffic here (for example, creating a CNAME record for `mailape.example.com` to redirect traffic to our ELB).

Let's look at how to SSH into the EC2 instances that our ASG launched.

SSHing into a Mail Ape EC2 Instance

The AWS CLI gives us many ways of getting information about our EC2 Instances. Let's find the address of our launched EC2 instance:

```
$ aws ec2 describe-instances \
 --region=us-west-2 \
 --filters='Name=tag:aws:cloudformation:stack-name,Values=mail_ape_1_0'
```

The `aws ec2 describe-instances` command will return a lot of information about all our EC2 instances. We can use the `--filters` command to restrict the EC2 instances returned. When we create a stack, many of the related resources are tagged with the stack name. This lets us filter for only those EC2 instances in our `mail_ape_1_0` stack.

The following is a (much) shortened version of the output:

```
{
  "Reservations": [
    {
      "Groups": [],
      "Instances": [
        {
          "ImageId": "ami-XXX",
          "InstanceId": "i-XXX",
          "InstanceType": "t2.micro",
          "KeyName": "mail_ape_production",
          "PublicDnsName": "ec2-XXX-XXX-XXX-XXX.us-
west-2.compute.amazonaws.com",
          "PublicIpAddress": "XXX",
          "State": {
            "Name": "running"
          },
          "IamInstanceProfile": {
```

```
            "Arn": "arn:aws:iam::XXX:instance-profile/infrastructure-
SQSClientInstance-XXX"
          },
          "SecurityGroups": [
            {
              "GroupName": "ssh-access"
            },
            {
              "GroupName": "web-access"
            }
          ],
          "Tags": [
            {
              "Key": "aws:cloudformation:stack-name",
              "Value": "mail_ape_1_0"
            } ] } ] } ] }
```

In the preceding output, note the `PublicDnsName` and the `KeyName`. Since we created that key earlier in this chapter, we can SSH into this instance:

```
$ ssh -i /path/to/saved/ssh/key ubuntu@ec2-XXX-XXX-XXX-XXX.us-
west-2.compute.amazonaws.com
```

Remember that the `XXX` you see in the preceding output will be replaced by real values in your system.

Now that we can SSH into the system, we can create and migrate our database.

Creating and migrating our database

For our first release we first need to create our database. To create our database we're going to create a script in `database/make_database.sh`:

```
#!/usr/bin/env bash

psql -v ON_ERROR_STOP=1 postgresql://$USER:$PASSWORD@$HOST/postgres <<-
EOSQL
    CREATE DATABASE mailape;
    CREATE USER mailape;
    GRANT ALL ON DATABASE mailape to "mailape";
    ALTER USER mailape PASSWORD '$DJANGO_DB_PASSWORD';
    ALTER USER mailape CREATEDB;
EOSQL
```

This script uses three variables from its environment:

- `$USER`: The Postgres master user username. We defined this as `master` in our `cloudformation/infrastructure.yaml`.
- `$PASSWORD`: The Postgres master user's password. We provided this as a parameter for when we created the `infrastructure` stack.
- `$DJANGO_DB_PASSWORD`: This is the password for the Django database. We provided this as a parameter to Packer when creating our AMI.

Next, we'll execute this script locally by providing the values as variables:

```
$ export USER=master
$ export PASSWORD=...
$ export DJANGO_DB_PASSWORD=...
$ bash database/make_database.sh
```

Our Mail Ape database is now created.

Next, let's SSH into our new EC2 instance and run our database migrations:

```
$ ssh -i /path/to/saved/ssh/key ubuntu@ec2-XXX-XXX-XXX-XXX.us-
west-2.compute.amazonaws.com
$ source /mailape/virtualenv/bin/activate
$ cd /mailape/django
$ export DJANGO_DB_NAME=mailape
$ export DJANGO_DB_USER=mailape
$ export DJANGO_DB_PASSWORD=...
$ export DJANGO_DB_HOST=XXX.XXX.us-west-2.rds.amazonaws.com
$ export DJANGO_DB_PORT=5432
$ export DJANGO_LOG_FILE=/var/log/mailape/mailape.log
$ export DJANGO_SECRET_KEY=...
$ export DJANGO_SETTINGS_MODULE=config.production_settings
$ python manage.py migrate
```

Our `manage.py migrate` command is very similar to what we've used in previous chapters. The main difference here is that we needed to SSH into our production EC2 instance first.

When `migrate` returns success, our database is ready and we're good to release our app.

Releasing Mail Ape 1.0

Now that we've migrated our database, we're ready to update the DNS records of `mailape.example.com` to point to our ELB's DNS records. Once the DNS records propagate, Mail Ape will be live.

Congratulations!

Scaling up and down with update-stack

One of the great things about using CloudFormation and Auto Scaling Groups is that it's easy to scale our system up and down. In this section, let's update our system to use two EC2 instances running Mail Ape.

We can update our CloudFormation template in `cloudformation/web_worker.yaml`:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Mail Ape web worker
Parameters:
  ..
Resources:
  LoadBalancer:
    ...
  LaunchConfig:
    ...
  WorkerGroup:
    Type: "AWS::AutoScaling::AutoScalingGroup"
    Properties:
      LaunchConfigurationName: !Ref LaunchConfig
      MinSize: 1
      MaxSize: 3
      DesiredCapacity: 2
      LoadBalancerNames:
        - !Ref LoadBalancer
Outputs:
  ..
```

We've updated our `DesiredCapacity` from 1 to 2. Now, instead of creating a new stack, let's update our existing stack:

```
$ aws cloudformation update-stack \
    --stack-name "mail_ape_1_0" \
    --template-body
"file:///path/to/mailape/cloudformation/web_worker.yaml" \
    --parameters \
      "ParameterKey=WorkerAMI,UsePreviousValue=true" \
      "ParameterKey=InstanceProfile,UsePreviousValue=true" \
    --region us-west-2
```

The preceding command looks much like our `create-stack` command. One convenient difference is that we don't need to provide the parameter values again—we can simply inform `UsePreviousValue=true` to tell AWS to reuse the same values as before.

Again, `describe-stack` will tell us when our update is complete:

```
aws cloudformation describe-stacks \
    --stack-name mail_ape_1_0 \
    --region us-west-2
```

The result is a large JSON object—here is a truncated example version:

```
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-
west-2:XXXX:stack/mail_ape_1_0/XXX",
            "StackName": "mail_ape_1_0",
            "Description": "Mail Ape web worker",
            "StackStatus": "UPDATE_COMPLETE"
        }
    ]
}
```

Once our `StackStatus` is `UPDATE_COMPLETE`, our ASG will be updated with a new setting. It can take a couple minutes for the ASG to launch the new EC2 instance, but we can use our previously created `describe-instances` command to look for it:

```
$ aws ec2 describe-instances \
 --region=us-west-2 \
 --filters='Name=tag:aws:cloudformation:stack-name,Values=mail_ape_1_0'
```

Eventually, it will return two instances. Here's a highly truncated version of what that output will look like:

```
{
  "Reservations": [
    {
      "Groups": [],
      "Instances": [
        {
          "ImageId": "ami-XXX",
          "InstanceId": "i-XXX",
          "PublicDnsName": "ec2-XXX-XXX-XXX-XXX.us-
west-2.compute.amazonaws.com",
          "State": { "Name": "running" }
        },
        {
          "ImageId": "ami-XXX",
          "InstanceId": "i-XXX",
          "PublicDnsName": "ec2-XXX-XXX-XXX-XXX.us-
west-2.compute.amazonaws.com",
          "State": { "Name": "running" }
        } ] } ] }
```

To scale down to one instance, just update your `web_worker.yaml` template and run `update-stack` again.

Congratulations! You now know how to scale Mail Ape up to handle a higher load and then scale back down during off peak periods.

Remember that Amazon charges are based on usage. If you scaled up as part of working through this book, remember to scale back down or you may be charged more than you expect. Ensure that you read up on the limits of the AWS free tier on `https://aws.amazon.com/free/`.

Summary

In this chapter, we've taken our Mail Ape app and launched it into a production environment in the AWS Cloud. We've used AWS CloudFormation to declare our AWS resources as code, making it as easy to track what we need and what changed as in the rest of our code base. We've built the image of our Mail Ape servers run using Packer, again giving us the ability to track our server configuration as code. Finally, we launched Mail Ape into the cloud and learned how to scale it up and down.

Now that we've come to the end of our journey learning to build Django web applications, let's review some of what we've learned. Over three projects we've seen how Django organizes code into models, views, and templates. We've learned how to do input validation with Django's form class and with Django Rest Framework's Serializer classes. We've examined security best practices, caching, and how to send emails. We've seen how to take our code and deploy into Linux servers, Docker containers, and the AWS Cloud.

You're ready to take your idea and launch it with Django! Go for it!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Django RESTful Web Services
Gastón C. Hillar

ISBN: 978-1-78883-392-9

- The best way to build a RESTful Web Service or API with Django and the Django REST Framework
- Develop complex RESTful APIs from scratch with Django and the Django REST Framework
- Work with either SQL or NoSQL data sources
- Design RESTful Web Services based on application requirements
- Use third-party packages and extensions to perform common tasks
- Create automated tests for RESTful web services
- Debug, test, and profile RESTful web services with Django and the Django REST Framework

Web Development with Django Cookbook
Aidas Bendoraitis

ISBN: 978-1-78328-689-8

- Get started with the basic configuration necessary to start any Django project
- Build a database structure out of reusable model mixins
- Manage forms and views and get to know some useful patterns that are used to create them
- Create handy template filters and tags that you can reuse in every project
- Integrate your own functionality into the Django CMS
- Manage hierarchical structures with MPTT
- Import data from local sources and external web services as well as exporting your data to third parties
- Implement a multilingual search with Haystack
- Test and deploy your project efficiently

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

H

I

L

M

O

P

Q

R

S

T

U

V

W