Building a basic blog using Django Framework

Installing Django

If you have already installed Django, you can skip this section and jump directly to the *Creating your first project* section. Django comes as a Python package and thus can be installed in any Python environment. If you haven't installed Django yet, the following is a quick guide to install Django for local development.

Django 2.0 requires Python version 3.4 or higher. In the examples for this book, we will use Python 3.6.5. If you're using Linux or macOS X, you probably have Python installed. If you are using Windows, you can download a Python installer at `https://www.python.org/downloads/windows/`.

If you are not sure whether Python is installed on your computer, you can verify it by typing `python` in the shell. If you see something like the following, then Python is installed on your computer:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If your installed Python version is lower than 3.4, or if Python is not installed on your computer, download Python 3.6.5 from `https://www.python.org/downloads/` and install it.

Since you will use Python 3, you don't have to install a database. This Python version comes with a built-in SQLite database. SQLite is a lightweight database that you can use with Django for development. If you plan to deploy your application in a production environment, you should use an advanced database, such as PostgreSQL, MySQL, or Oracle. You can get more information about how to get your database running with Django at `https://docs.djangoproject.com/en/2.0/topics/install/#database-installation`.

Creating an isolated Python environment

It is recommended that you use `virtualenv` to create isolated Python environments, so that you can use different package versions for different projects, which is far more practical than installing Python packages system-wide. Another advantage of using `virtualenv` is that you won't need any administration privileges to install Python packages. Run the following command in your shell to install `virtualenv`:

```
pip install virtualenv
```

After you install `virtualenv`, create an isolated environment with the following command:

```
virtualenv my_env
```

This will create a `my_env/` directory, including your Python environment. Any Python libraries you install while your virtual environment is active will go into the `my_env/lib/python3.6/site-packages` directory.

> *If your system comes with Python 2.X and you have installed Python 3.X, you have to tell `virtualenv` to use the latter.*

You can locate the path where Python 3 is installed and use it to create the virtual environment with the following commands:

```
zenx$ which python3
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3
zenx$ virtualenv my_env -p
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3
```

Run the following command to activate your virtual environment:

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

The shell prompt will include the name of the active virtual environment enclosed in parentheses, as follows:

```
(my_env)laptop:~ zenx$
```

You can deactivate your environment at any time with the `deactivate` command.

You can find more information about `virtualenv` at `https://virtualenv.pypa.io/en/latest/`.

On top of `virtualenv`, you can use `virtualenvwrapper`. This tool provides wrappers that make it easier to create and manage your virtual environments. You can download it from `https://virtualenvwrapper.readthedocs.io/en/latest/`.

Installing Django with pip

The `pip` package management system is the preferred method for installing Django. Python 3.6 comes with `pip` preinstalled, but you can find `pip` installation instructions at `https://pip.pypa.io/en/stable/installing/`.

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

```
pip install Django==2.0.5
```

Django will be installed in the Python `site-packages/` directory of your virtual environment.

Now, check whether Django has been successfully installed. Run `python` on a terminal, import Django, and check its version, as follows:

```
>>> import django
>>> django.get_version()
'2.0.5'
```

If you get the preceding output, Django has been successfully installed on your machine.

Django can be installed in several other ways. You can find a complete installation guide at `https://docs.djangoproject.com/en/2.0/topics/install/`.

Creating your first project

Our first Django project will be building a complete blog. Django provides a command that allows you to create an initial project file structure. Run the following command from your shell:

```
django-admin startproject mysite
```

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

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

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

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

These files are as follows:

- `manage.py`: This is a command-line utility used to interact with your project. It is a thin wrapper around the `django-admin.py` tool. You don't need to edit this file.
- `mysite/`: This is your project directory, which consists of the following files:
    - `__init__.py`: An empty file that tells Python to treat the

`mysite` directory as a Python module.

- `settings.py`: This indicates settings and configuration for your project and contains initial default settings.
- `urls.py`: This is the place where your URL patterns live. Each URL defined here is mapped to a view.
- `wsgi.py`: This is the configuration to run your project as a **Web Server Gateway Interface** (**WSGI**) application.

The generated `settings.py` file contains the project settings, including a basic configuration to use an SQLite 3 database and a list named `INSTALLED_APPS`, which contains common Django applications that are added to your project by default. We will go through these applications later in the *Project settings* section.

To complete the project setup, we will need to create the tables in the database required by the applications listed in `INSTALLED_APPS`. Open the shell and run the following commands:

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

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

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying 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 sessions.0001_initial... OK
```

The preceding lines are the database migrations that are applied by Django. By applying migrations, the tables for the initial applications are created in the database. You will learn about the `migrate` management command in the *Creating and applying migrations* section of this chapter.

Running the development server

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

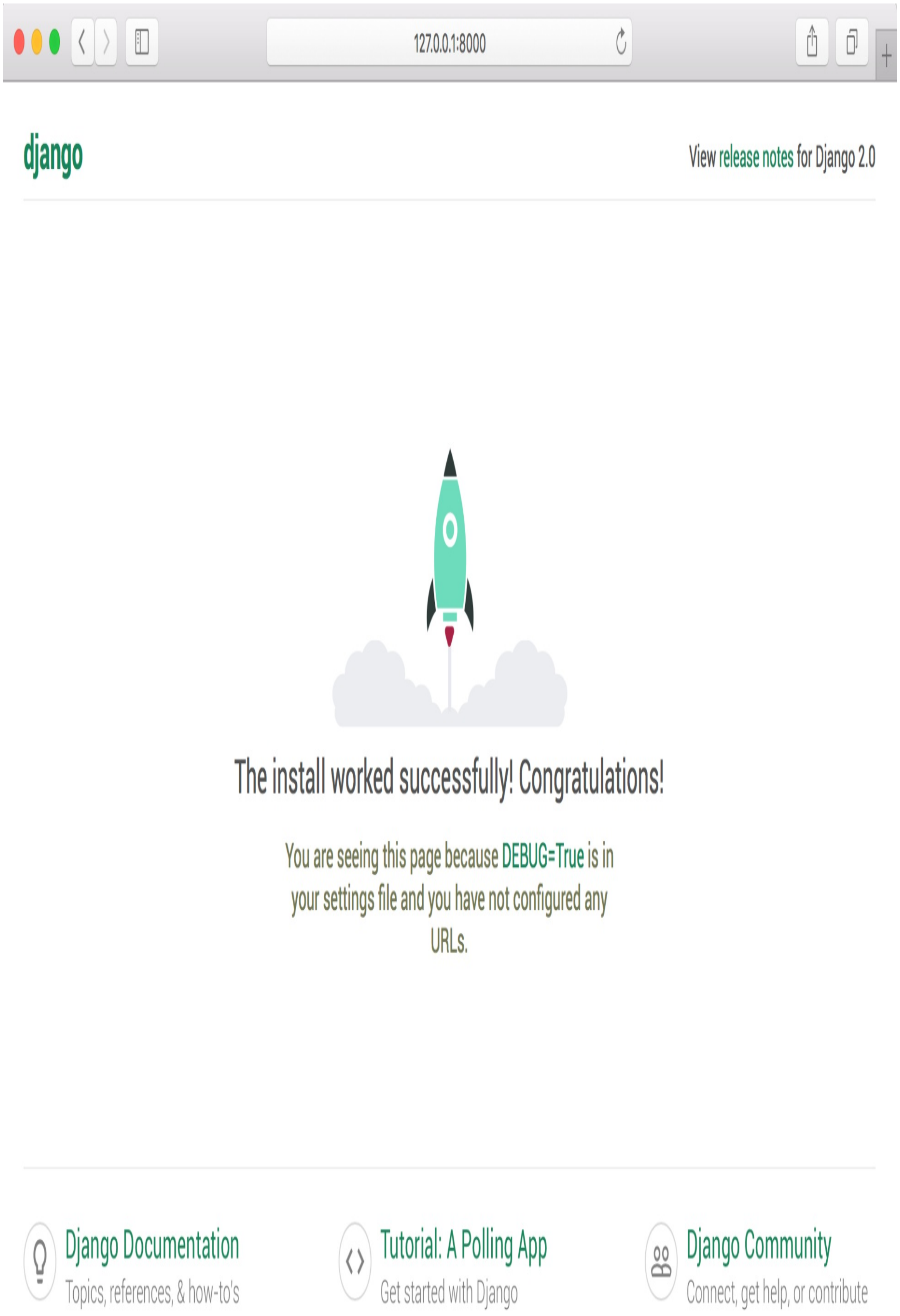
Start the development server by typing the following command from your project's root folder:

```
python manage.py runserver
```

You should see something like this:

```
Performing system checks...

System check identified no issues (0 silenced).
May 06, 2018 - 17:17:31
Django version 2.0.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

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


127.0.0.1:8000

django

View release notes for Django 2.0

The install worked successfully! Congratulations!

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

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

Tutorial: A Polling App
Get started with Django

Django Community
Connect, get help, or contribute

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

```
[06/May/2018 17:20:30] "GET / HTTP/1.1" 200 16348
```

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

You can indicate Django to run the development server on a custom host and port or tell it to run your project, loading a different settings file, as follows:

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

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

Remember that this server is only intended for development and is not suitable for production use. In order to deploy Django in a production environment, you should run it as a WSGI application using a real web server, such as Apache, Gunicorn, or uWSGI. You can find more information on how to deploy Django with different web servers at `https://docs.djangoproject.com/en/2.0/howto/deployment/wsgi/`.

`Chapter 13`, *Going Live*, explains how to set up a production environment for your Django projects.

Project settings

Let's open the `settings.py` file and take a look at the configuration of our project. There are several settings that Django includes in this file, but these are only a part of all the Django settings available. You can see all settings and their default values in `https://docs.djangoproject.com/en/2.0/ref/settings/`.

The following settings are worth looking at:

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

    - `django.contrib.auth`: An authentication framework
    - `django.contrib.contenttypes`: A framework for handling content types
    - `django.contrib.sessions`: A session framework
    - `django.contrib.messages`: A messaging framework
    - `django.contrib.staticfiles`: A framework for managing static files

- `MIDDLEWARE` is a list that contains middleware to be executed.
- `ROOT_URLCONF` indicates the Python module where the root URL patterns of your application are defined.
- `DATABASES` is a dictionary that contains the settings for all the databases to be used in the project. There must always be a default database. The default configuration uses an SQLite3 database.
- `LANGUAGE_CODE` defines the default language code for this Django site.
- `USE_TZ` tells Django to activate/deactivate timezone support. Django comes with support for timezone-aware datetime. This setting is set to `True` when you create a new project using the `startproject` management command.

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

Projects and applications

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

Creating an application

Now, let's create our first Django application. We will create a blog application from scratch. From the project's root directory, run the following command:

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

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

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

These files are as follows:

- `admin.py`: This is where you register models to include them in the Django administration site—using the Django admin site is optional.
- `apps.py`: This includes the main configuration of the `blog` application.
- `migrations`: This directory will contain database migrations of your application. Migrations allow Django to track your model changes and synchronize the database accordingly.

- `models.py`: Data models of your application—all Django applications need to have a `models.py` file, but this file can be left empty.
- `tests.py`: This is where you can add tests for your application.
- `views.py`: The logic of your application goes here; each view receives an HTTP request, processes it, and returns a response.

Designing the blog data schema

We will start designing our blog data schema by defining the data models for our blog. A model is a Python class that subclasses `django.db.models.Model`, in which each attribute represents a database field. Django will create a table for each model defined in the `models.py` file. When you create a model, Django provides you with a practical API to query objects in the database easily.

First, we will define a `Post` model. Add the following lines to the `models.py` file of the `blog` application:

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

class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
                              choices=STATUS_CHOICES,
                              default='draft')

    class Meta:
        ordering = ('-publish',)
    def __str__(self):
        return self.title
```

This is our data model for blog posts. Let's take a look at the fields we just defined for this model:

- `title`: This is the field for the post title. This field is `CharField`, which translates into a `VARCHAR` column in the SQL database.
- `slug`: This is a field intended to be used in URLs. A slug is a short label that contains only letters, numbers, underscores, or hyphens. We will use the `slug` field to build beautiful, SEO-friendly URLs for our blog posts. We have added the `unique_for_date` parameter to this field so that we can build URLs for posts using their publish date and `slug`. Django will prevent multiple posts from having the same `slug` for a given date.
- `author`: This field is a foreign key. It defines a many-to-one relationship. We are telling Django that each post is written by a user, and a user can write any number of posts. For this field, Django will create a foreign key in the database using the primary key of the related model. In this case, we are relying on the `User` model of the Django authentication system. The `on_delete` parameter specifies the behavior to adopt when the referenced object is deleted. This is not specific to Django; it is an SQL standard. Using `CASCADE`, we specify that when the referenced user is deleted, the database will also delete its related blog posts. You can take a look at all possible options at `https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.ForeignKey.on_delete`. We specify

the name of the reverse relationship, from `User` to `Post`, with the `related_name` attribute. This will allow us to access related objects easily. We will learn more about this later.

- `body`: This is the body of the post. This field is a text field, which translates into a `TEXT` column in the SQL database.
- `publish`: This datetime indicates when the post was published. We use Django's timezone `now` method as the default value. This returns the current datetime in a timezone-aware format. You can think of it as a timezone-aware version of the standard Python `datetime.now` method.
- `created`: This datetime indicates when the post was created. Since we are using `auto_now_add` here, the date will be saved automatically when creating an object.
- `updated`: This datetime indicates the last time the post was updated. Since we are using `auto_now` here, the date will be updated automatically when saving an object.
- `status`: This field shows the status of a post. We use a `choices` parameter, so the value of this field can only be set to one of the given choices.

Django comes with different types of fields that you can use to define your models. You can find all field types at `https://docs.djangoproject.com/en/2.0/ref/models/fields/`.

The `Meta` class inside the model contains metadata. We tell Django to sort results in the `publish` field in descending order by default when we query the database. We specify descending order using the negative prefix. By doing so, posts published recently will appear first.

The `__str__()` method is the default human-readable representation of the object. Django will use it in many places, such as the administration site.

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

Activating your application

In order for Django to keep track of our application and be able to create database tables for its models, we have to activate it. To do this, edit the `settings.py` file and add `blog.apps.BlogConfig` to the `INSTALLED_APPS` setting. It should look like this:

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

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

Creating and applying migrations

Now that we have a data model for our blog posts, we will need a database table for it. Django comes with a migration system that tracks the changes done to models and allows to propagate them into the database. The `migrate` command applies migrations for all applications listed in `INSTALLED_APPS`; it synchronizes the database with the current models and existing migrations.

First, you will need to create an initial migration for our `Post` model. In the root directory of your project, run the following command:

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

You should get the following output:

```
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model Post
```

Django just created the `0001_initial.py` file inside the `migrations` directory of the `blog` application. You can open that file to see how a migration appears. A migration specifies dependencies on other migrations and operations to perform in the database to synchronize it with model changes.

Let's take a look at the SQL code that Django will execute in the database to create the table for our model. The `sqlmigrate` command takes migration names and returns their SQL without executing it. Run the following command to inspect the SQL output of our first migration:

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

The output should look as follows:

```
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"title" varchar(250) NOT NULL, "slug" varchar(250) NOT NULL, "body" text NOT
NULL, "publish" datetime NOT NULL, "created" datetime NOT NULL, "updated"
datetime NOT NULL, "status" varchar(10) NOT NULL, "author_id" integer NOT
NULL REFERENCES "auth_user" ("id"));
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

The exact output depends on the database you are using. The preceding output is generated for SQLite. As you can see in the preceding output, Django generates the table names by combining the app name and the lowercase name of the model (`blog_post`), but you can also specify a custom database name for your model in the `Meta` class of the model using the `db_table` attribute. Django creates a primary key automatically for each model, but you can also override this by specifying `primary_key=True` in one of your model fields. The default primary key is an `id` column, which consists of an integer that is incremented automatically. This column corresponds to the `id` field that is automatically added to your models.

Let's sync our database with the new model. Run the following command to apply existing migrations:

```
python manage.py migrate
```

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

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

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

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

Creating an administration site for your models

Now that we have defined the `Post` model, we will create a simple administration site to manage your blog posts. Django comes with a built-in administration interface that is very useful for editing content. The Django admin site is built dynamically by reading your model metadata and providing a production-ready interface for editing content. You can use it out of the box, configuring how you want your models to be displayed in it.

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

Creating a superuser

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

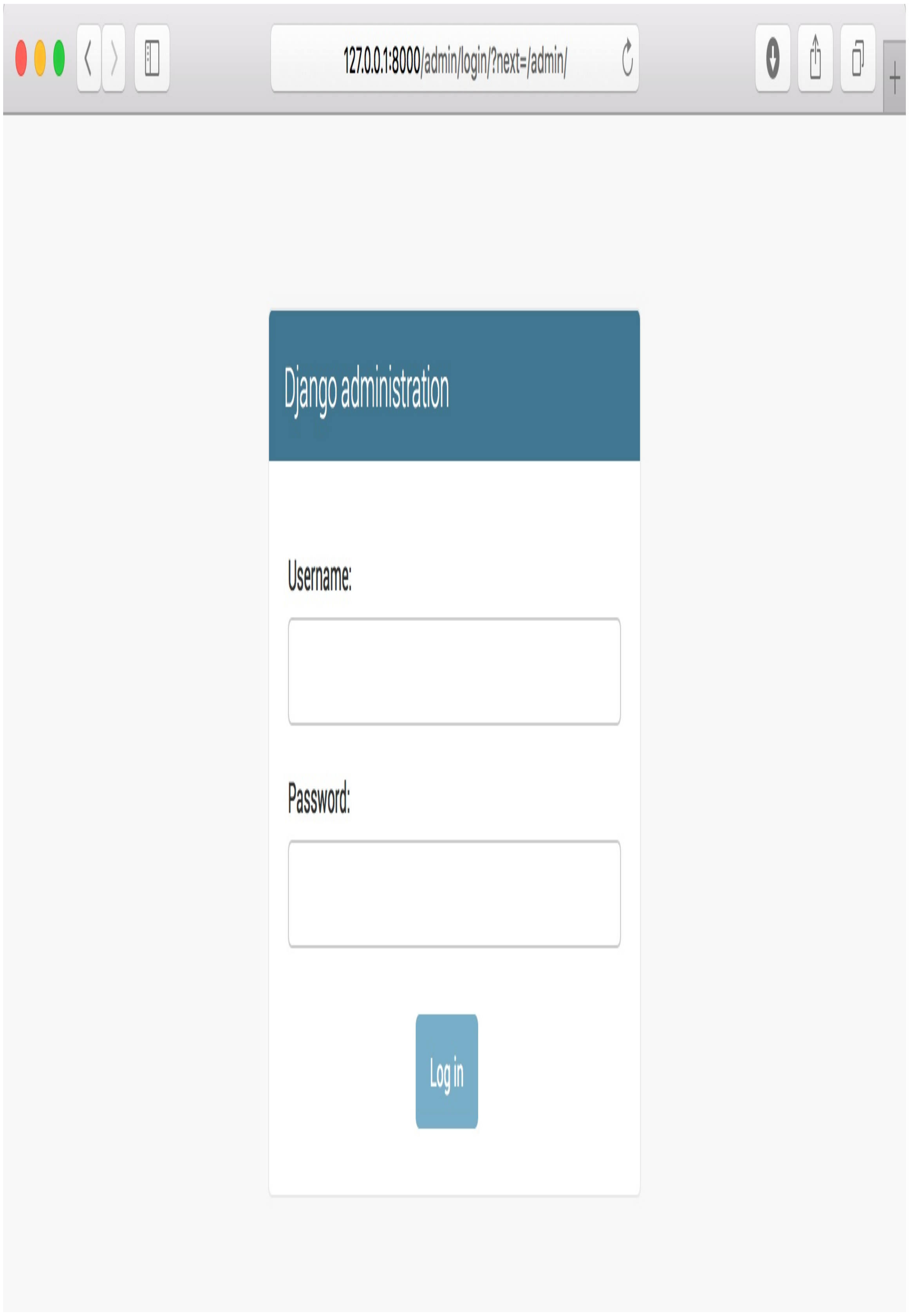
```
python manage.py createsuperuser
```

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

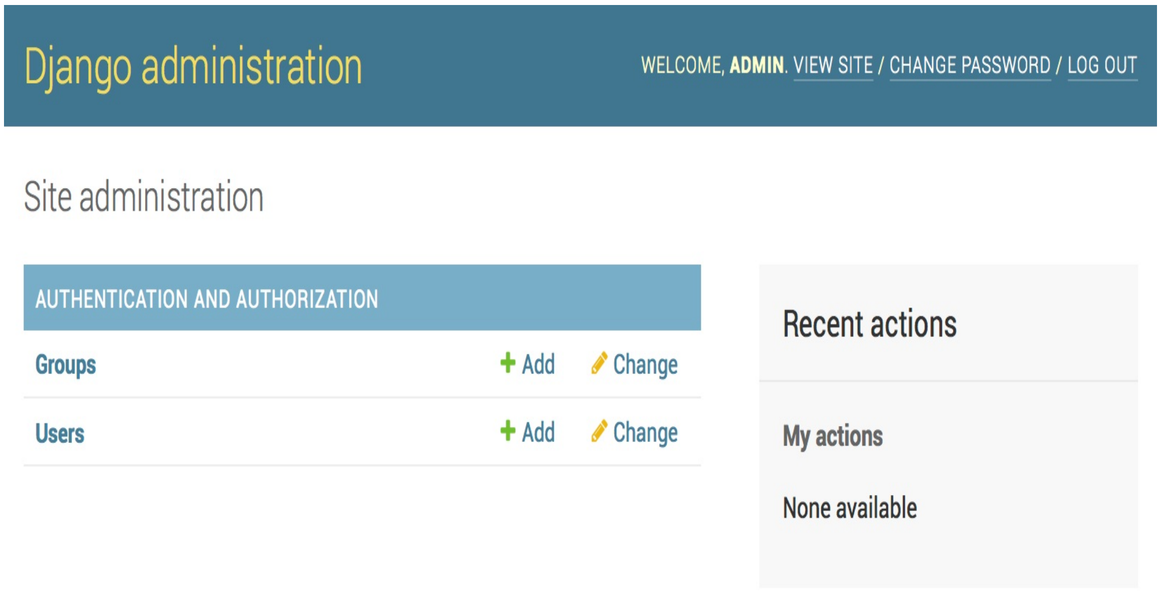
```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
Superuser created successfully.
```

The Django administration site

Now, start the development server with the `python manage.py runserver` command and open `http://127.0.0.1:8000/admin/` in your browser. You should see the administration login page, as shown in the following screenshot:


127.0.0.1:8000/admin/login/?next=/admin/
Django administration
Username:
Password:
Log in

Log in using the credentials of the user you created in the preceding step. You will see the admin site index page, as shown in the following screenshot:



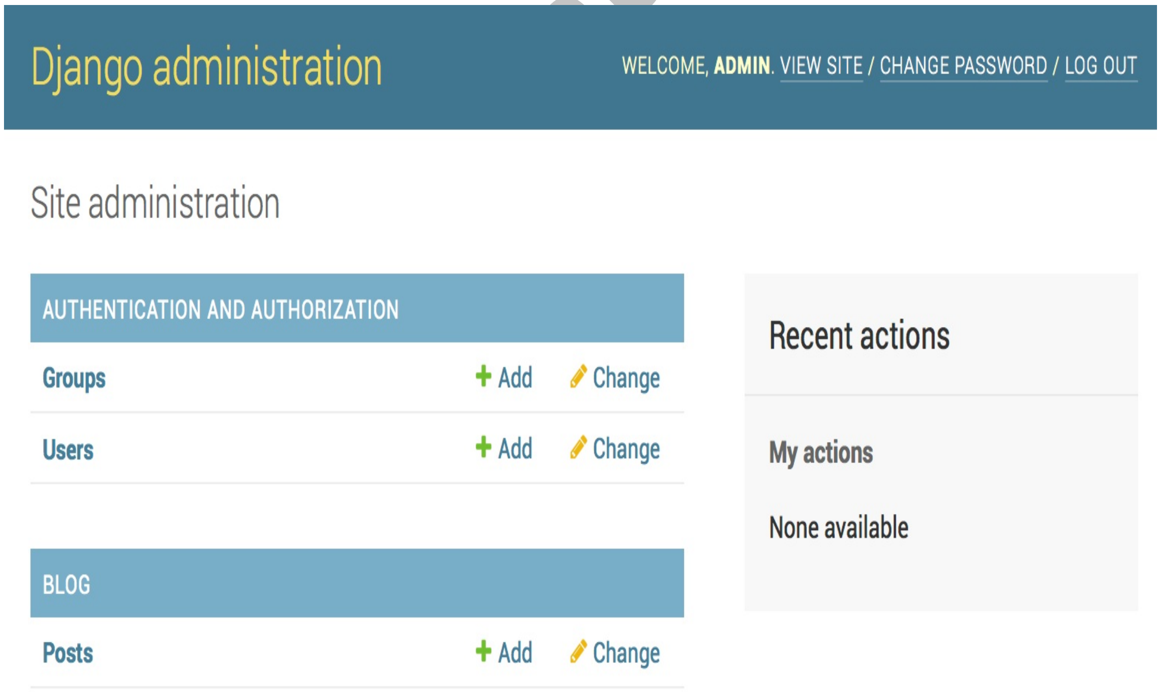
The `Group` and `User` models you see in the preceding screenshot are part of the Django authentication framework located in `django.contrib.auth`. If you click on Users, you will see the user you created previously. The `Post` model of your `blog` application has a relationship with this `User` model. Remember that it is a relationship defined by the `author` field.

Adding your models to the administration site

Let's add your blog models to the administration site. Edit the `admin.py` file of your `blog` application and make it look like this:

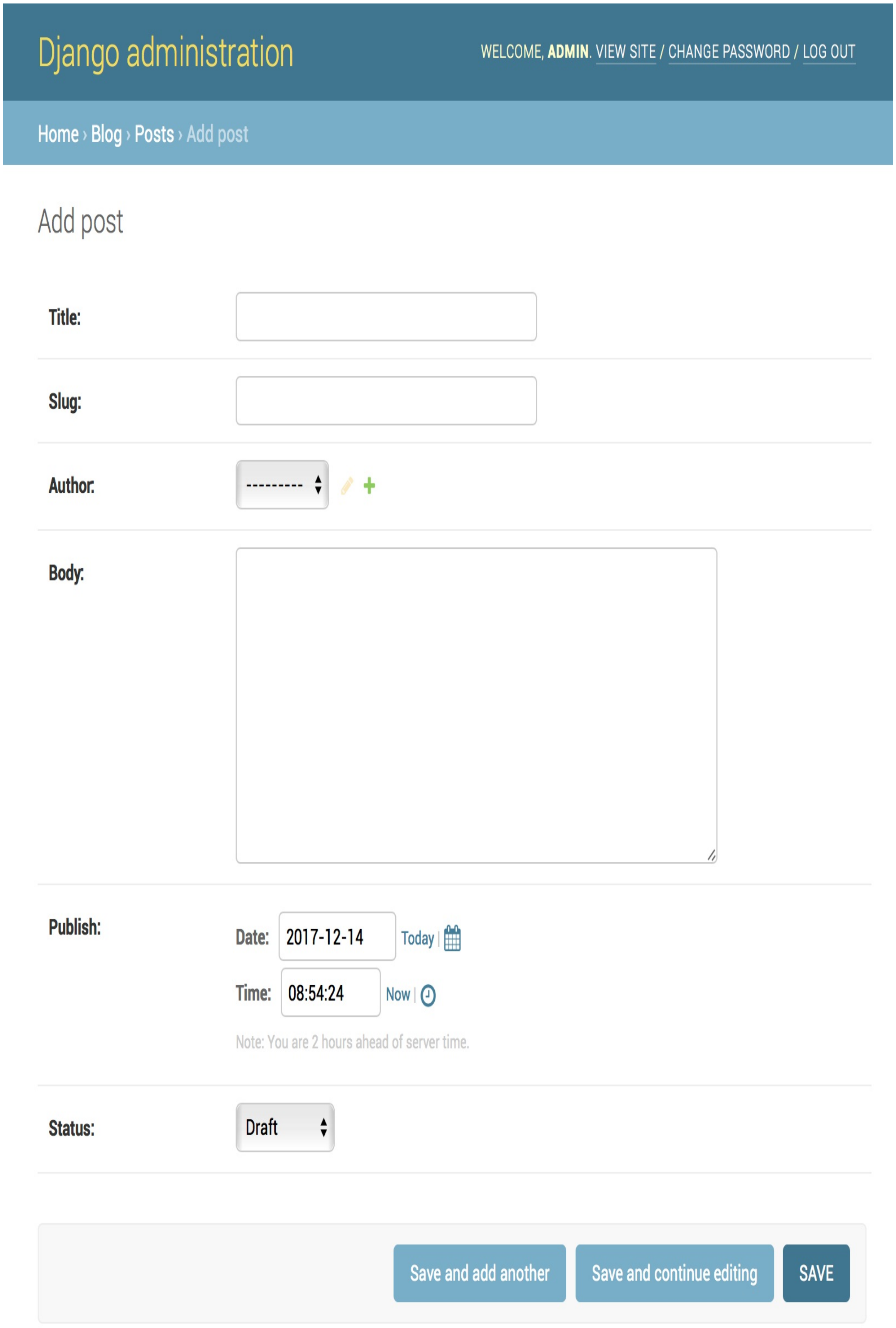
```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

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



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

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


Django administration

WELCOME, **ADMIN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

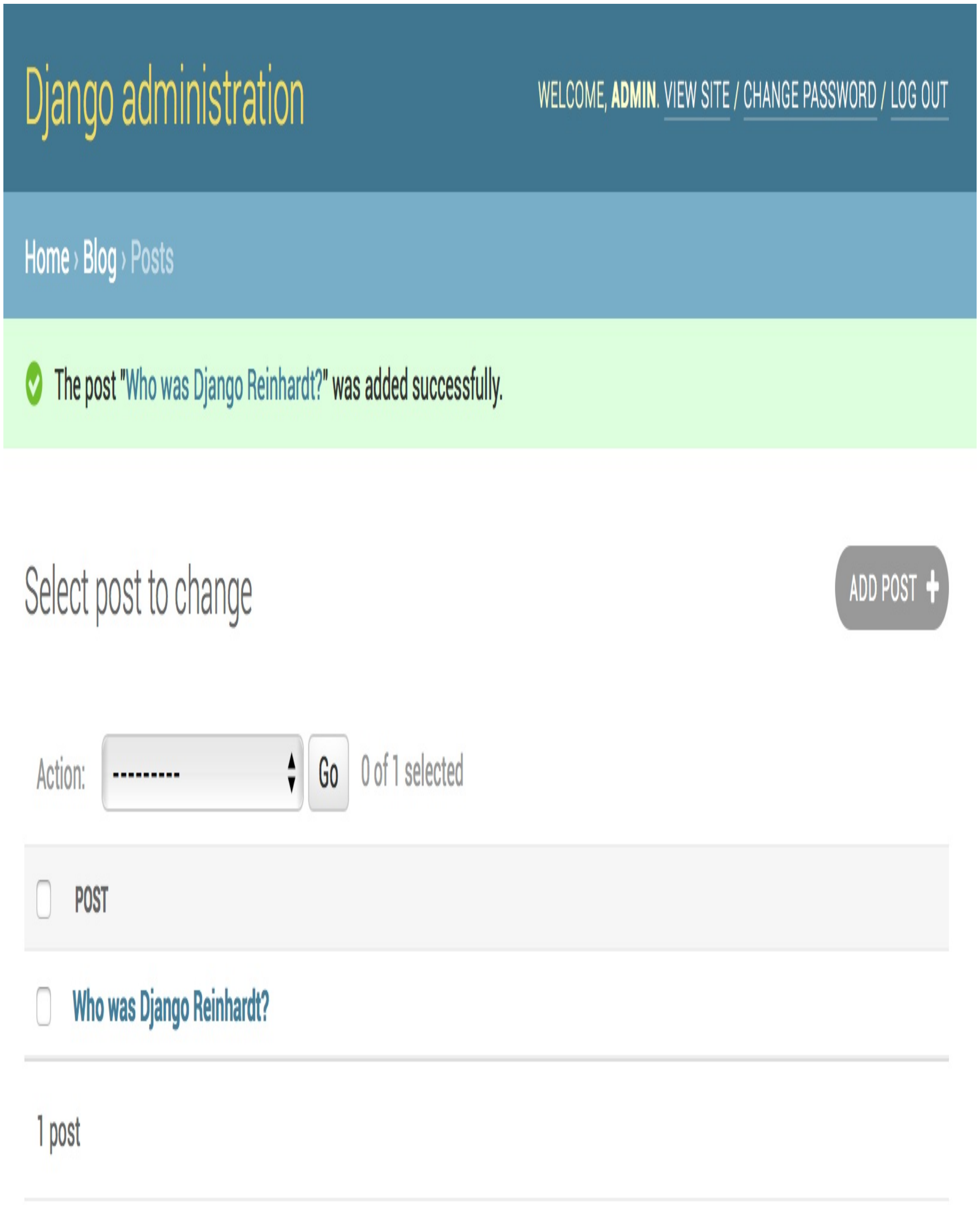
Home › Blog › Posts › Add post

Add post

Title:	
Slug:	
Author:	---------
Body:	
Publish:	Date: 2017-12-14 Today Time: 08:54:24 Now Note: You are 2 hours ahead of server time.
Status:	Draft

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

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



Django administration

WELCOME, **ADMIN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Blog › Posts

The post "Who was Django Reinhardt?" was added successfully.

Select post to change

0 of 1 selected

	POST
☐	Who was Django Reinhardt?

1 post

Customizing the way models are displayed

Now, we will take a look at how to customize the admin site. Edit the `admin.py` file of your blog application and change it, as follows:

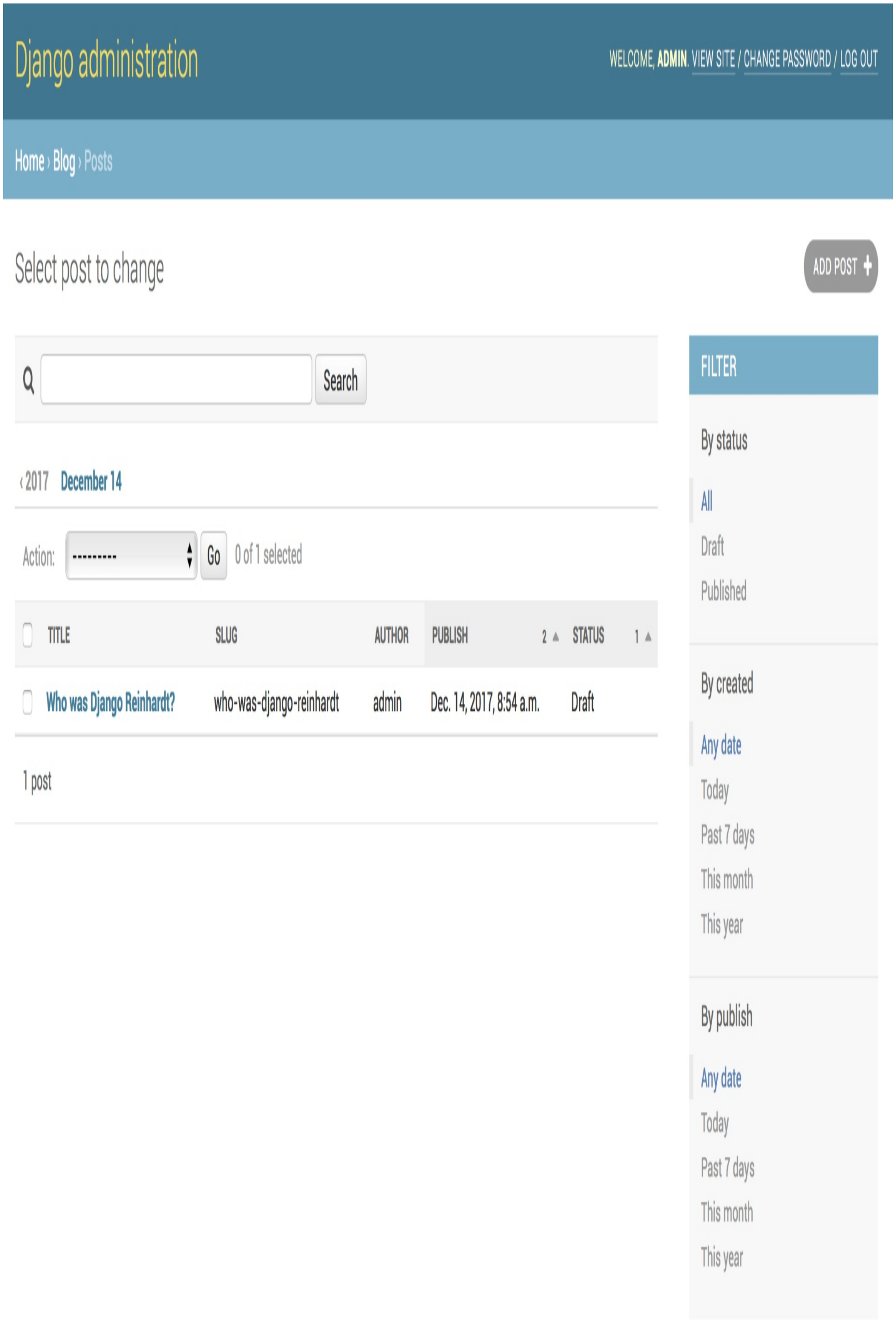
```
from django.contrib import admin
from .models import Post

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

We are telling the Django admin site that our model is registered in the admin site using a custom class that inherits from `ModelAdmin`. In this class, we can include information about how to display the model in the admin site and how to interact with it. The `list_display` attribute allows you to set the fields of your model that you want to display in the admin object list page. The `@admin.register()` decorator performs the same function as the `admin.site.register()` function we have replaced, registering the `ModelAdmin` class that it decorates.

Let's customize the admin model with some more options, using the following code:

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

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



Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Blog › Posts

Select post to change

ADD POST

Search

‹ 2017 December 14

Action: --------- Go 0 of 1 selected

	TITLE	SLUG	AUTHOR	PUBLISH 2 ▲	STATUS 1 ▲
☐	Who was Django Reinhardt?	who-was-django-reinhardt	admin	Dec. 14, 2017, 8:54 a.m.	Draft

1 post

FILTER

By status

All

Draft

Published

By created

Any date

Today

Past 7 days

This month

This year

By publish

Any date

Today

Past 7 days

This month

This year

You can see that the fields displayed on the post list page are the ones you specified in the `list_display` attribute. The list page now includes a right sidebar that allows you to filter the results by the fields included in the `list_filter` attribute. A Search bar has appeared on the page. This is because we have defined a list of searchable fields using the `search_fields` attribute. Just below the Search bar, there are navigation links to navigate through a date hierarchy: this has been defined by the `date_hierarchy` attribute. You can also see that the posts are ordered by Status and Publish columns by default. We have specified the default order using the `ordering` attribute.

Now, click on the Add Post link. You will also note some changes here. As you type the title of a new post, the `slug` field is filled in automatically. We have told Django to prepopulate the `slug` field with the input of the `title` field using the `prepopulated_fields` attribute. Also, now, the `author` field is displayed with a lookup widget that can scale much better than a drop-down select input when you have thousands of users, as shown in the following screenshot:

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

Working with QuerySet and managers

Now that you have a fully functional administration site to manage your blog's content, it's time to learn how to retrieve information from the database and interact with it. Django comes with a powerful database abstraction API that lets you create, retrieve, update, and delete objects easily. The Django **Object-relational mapper** is compatible with MySQL, PostgreSQL, SQLite, and Oracle. Remember that you can define the database of your project in the `DATABASES` setting of your project's `settings.py` file. Django can work with multiple databases at a time, and you can program database routers to create custom routing schemes.

Once you have created your data models, Django gives you a free API to interact with them. You can find the data model reference of the official documentation at `https://docs.djangoproject.com/en/2.0/ref/models/`.

Creating objects

Open the terminal and run the following command to open the Python shell:

```
python manage.py shell
```

Then, type the following lines:

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

Let's analyze what this code does. First, we will retrieve the `user` object with the username `admin`:

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

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

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

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

This object is in memory and is not persisted to the database.

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

```
post.save()
```

The preceding action performs an `INSERT` SQL statement behind the scenes. We have seen how to create an object in memory first and then persist it to the database, but we can also create the object and persist it into the database in a single operation using the `create()` method, as follows:

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

Updating objects

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

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

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

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

Retrieving objects

The Django **object-relational mapping** (**ORM**) is based on QuerySets. A QuerySet is a collection of objects from your database that can have several filters to limit the results. You already know how to retrieve a single object from the database using the `get()` method. We have accessed this method using `Post.objects.get()`. Each Django model has at least one manager, and the default manager is called **objects**. You get a `QuerySet` object using your model manager. To retrieve all objects from a table, you just use the `all()` method on the default objects manager, like this:

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

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

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

Using the filter() method

To filter a QuerySet, you can use the `filter()` method of the manager. For example, we can retrieve all posts published in the year 2017 using the following QuerySet:

```
Post.objects.filter(publish__year=2017)
```

You can also filter by multiple fields. For example, we can retrieve all posts published in 2017 by the author with the username `admin`:

```
Post.objects.filter(publish__year=2017, author__username='admin')
```

This equates to building the same QuerySet chaining multiple filters:

```
Post.objects.filter(publish__year=2017) \
            .filter(author__username='admin')
```

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

Using exclude()

You can exclude certain results from your QuerySet using the `exclude()` method of the manager. For example, we can retrieve all posts published in 2017 whose titles don't start with `Why`:

```
Post.objects.filter(publish__year=2017) \
            .exclude(title__startswith='Why')
```

Using order_by()

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

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

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

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

Deleting objects

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

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

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

When QuerySets are evaluated

You can concatenate as many filters as you like to a QuerySet, and you will not hit the database until the QuerySet is evaluated. QuerySets are only evaluated in the following cases:

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

Creating model managers

As we previously mentioned, `objects` is the default manager of every model that retrieves all objects in the database. However, we can also define custom managers for our models. We will create a custom manager to retrieve all posts with the `published` status.

There are two ways to add managers to your models: you can add extra manager methods or modify initial manager QuerySets. The first method provides you with a QuerySet API such as `Post.objects.my_manager()`, and the latter provides you with `Post.my_manager.all()`. The manager will allow us to retrieve posts using `Post.published.all()`.

Edit the `models.py` file of your `blog` application to add the custom manager:

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super(PublishedManager,
                     self).get_queryset()\
                          .filter(status='published')

class Post(models.Model):
    # ...
    objects = models.Manager() # The default manager.
    published = PublishedManager() # Our custom manager.
```

The `get_queryset()` method of a manager returns the QuerySet that will be executed. We override this method to include our custom filter in the final QuerySet. We have defined our custom manager and added it to the `Post` model; we can now use it to perform queries. Let's test it.

Start the development server again with the following command:

```
python manage.py shell
```

Now, you can retrieve all published posts whose title starts with `Who` using the following command:

```
Post.published.filter(title__startswith='Who')
```

Building list and detail views

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

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

Creating list and detail views

Let's start by creating a view to display the list of posts. Edit the `views.py` file of your `blog` application and make it look like this:

```
from django.shortcuts import render, get_object_or_404
from .models import Post

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

You just created your first Django view. The `post_list` view takes the `request` object as the only parameter. Remember that this parameter is required by all views. In this view, we are retrieving all the posts with the `published` status using the `published` manager we created previously.

Finally, we are using the `render()` shortcut provided by Django to render the list of posts with the given template. This function takes the `request` object, the template path, and the context variables to render the given template. It returns an `HttpResponse` object with the rendered text (normally, HTML code). The `render()` shortcut takes the request context into account, so any variable set by template context processors is accessible by the given template. Template context processors are just callables that set variables into the context. You will learn how to use them in `Chapter 3`, *Extending Your Blog Application.*

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

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

This is the post detail view. This view takes `year`, `month`, `day`, and `post` parameters to retrieve a published post with the given slug and date. Note that when we created the `Post` model, we added the `unique_for_date` parameter to the `slug` field. This way, we ensure that there will be only one post with a slug for a given date, and thus, we can retrieve single posts using date and slug. In the detail view, we use the `get_object_or_404()` shortcut to retrieve the desired post. This function retrieves the object that matches the given parameters or launches an HTTP 404 (not found) exception if no object is found. Finally, we use the `render()` shortcut to render the retrieved post using a template.

Adding URL patterns for your views

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

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

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

app_name = 'blog'

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

In the preceding code, we define an application namespace with the `app_name` variable. This allows us to organize URLs by application and use the name when referring to them. We define two different patterns using the `path()` function. The first URL pattern doesn't take any arguments and is mapped to the `post_list` view. The second pattern takes the following four arguments and is mapped to the `post_detail` view:

- `year`: Requires an integer
- `month`: Requires an integer
- `day`: Requires an integer
- `post`: Can be composed of words and hyphens

We use angle brackets to capture the values from the URL. Any value specified in the URL pattern as `<parameter>` is captured as a string. We use path converters, such as `<int:year>`, to specifically match and return an integer and `<slug:post>` to specifically match a slug (a string consisting of ASCII letters or numbers, plus the hyphen and underscore characters). You can see all path converters provided by Django at `https://docs.djangoproject.com/en/2.0/topics/http/urls/#path-converters`.

If using `path()` and converters isn't sufficient for you, you can use `re_path()` instead to define complex URL patterns with Python regular expressions. You can learn more about defining URL patterns with regular expressions at `https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.re_path`. If you haven't worked with regular expressions before, you might want to take a look at the *Regular Expression HOWTO* located at `https://docs.python.org/3/howto/regex.html` first.

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

Now, you have to include the URL patterns of the `blog` application in the main URL patterns of the project. Edit the `urls.py` file located in the `mysite` directory of your project and make it look like the following:

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

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

The new URL pattern defined with `include` refers to the URL patterns defined in the blog application so that they are included under the `blog/` path. We include these patterns under the namespace `blog`. Namespaces have to be unique across your entire project. Later, we will refer to our blog URLs easily by including the namespace, building them, for example, `blog:post_list` and `blog:post_detail`. You can learn more about URL namespaces at https://docs.djangoproject.com/en/2.0/topics/http/urls/#url-namespaces.

Canonical URLs for models

You can use the `post_detail` URL that you have defined in the preceding section to build the canonical URL for `Post` objects. The convention in Django is to add a `get_absolute_url()` method to the model that returns the canonical URL of the object. For this method, we will use the `reverse()` method that allows you to build URLs by their name and passing optional parameters. Edit your `models.py` file and add the following:

```
from django.urls import reverse

class Post(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('blog:post_detail',
                       args=[self.publish.year,
                             self.publish.month,
                             self.publish.day,
                             self.slug])
```

We will use the `get_absolute_url()` method in our templates to link to specific posts.

Creating templates for your views

We have created views and URL patterns for the `blog` application. Now, it's time to add templates to display posts in a user-friendly manner.

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

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

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

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

- Template tags control the rendering of the template and look like `{% tag %}`.
- Template variables get replaced with values when the template is rendered and look like `{{ variable }}`.

- Template filters allow you to modify variables for display and look like `{{ variable|filter }}`.

You can see all built-in template tags and filters in https://docs.djangoproject.com/en/2.0/ref/templates/builtins/.

Let's edit the `base.html` file and add the following code:

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

`{% load static %}` tells Django to load the `static` template tags that are provided by the `django.contrib.staticfiles` application, which is contained in the `INSTALLED_APPS` setting. After loading it, you are able to use the `{% static %}` template filter throughout this template. With this template filter, you can include static files, such as the `blog.css` file, that you will find in the code of this example under the `static/` directory of the `blog` application. Copy the `static/` directory from the code that comes along with this chapter into the same location of your project to apply the CSS style sheets.

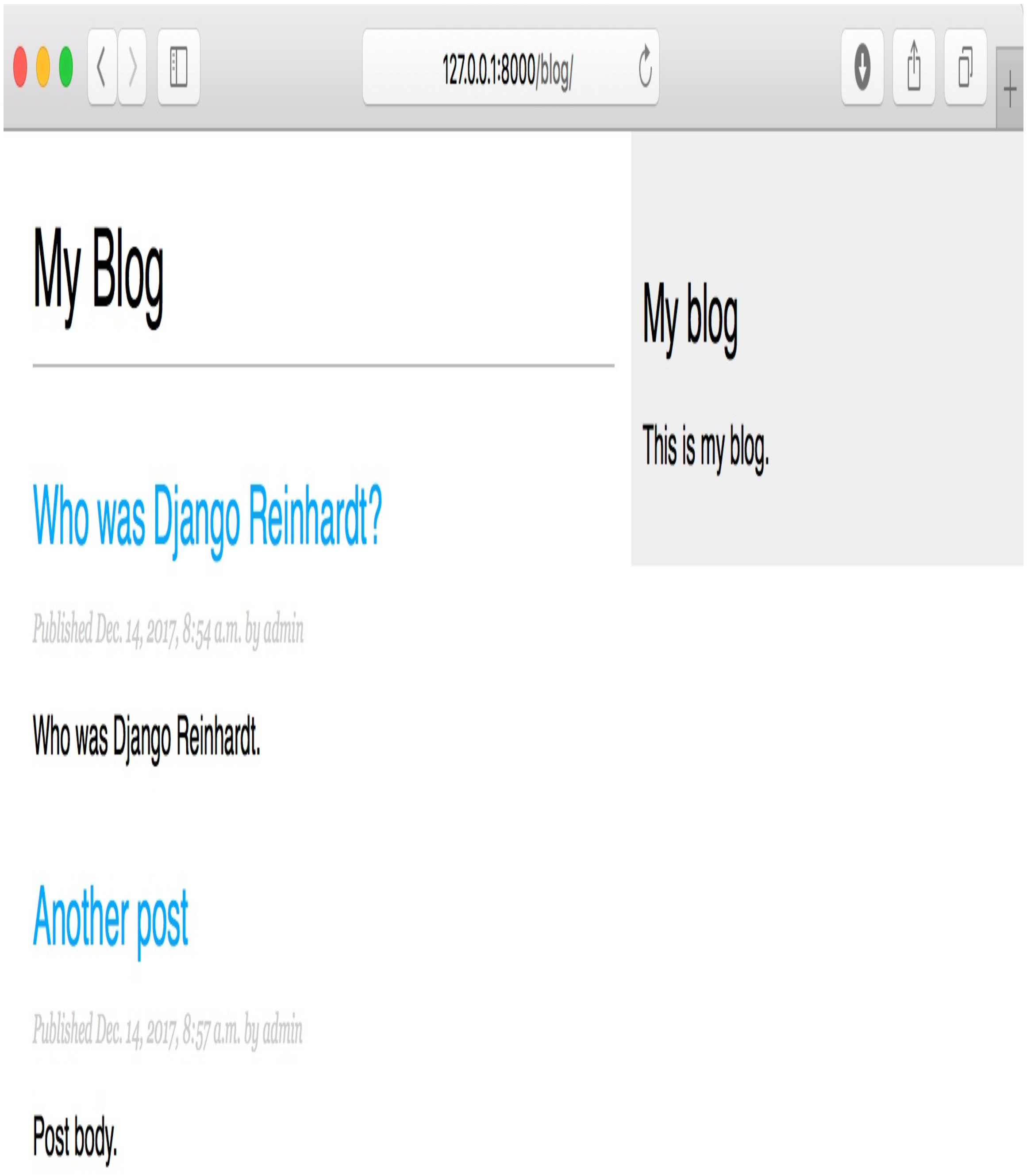
You can see that there are two `{% block %}` tags. These tell Django that we want to define a block in that area. Templates that inherit from this template can fill in the blocks with content. We have defined a block called `title` and a block called `content`.

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

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

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

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

With the `{% extends %}` template tag, we tell Django to inherit from the `blog/base.html` template. Then, we are filling the `title` and `content` blocks of the base template with content. We iterate through the posts and display their title, date, author, and body, including a link in the title to the canonical URL of the post. In the body of the post, we are applying two template filters: `truncatewords` truncates the value to the number of words specified, and `linebreaks` converts the output into HTML line breaks. You can concatenate as many template filters as you wish; each one will be applied to the output generated by the preceding one.

Open the shell and execute the `python manage.py runserver` command to start the development server. Open `http://127.0.0.1:8000/blog/` in your browser, and you will see everything running. Note that you need to have some posts with the Published status to show them here. You should see something like this:



My Blog

Who was Django Reinhardt?

Published Dec. 14, 2017, 8:54 a.m. by admin

Who was Django Reinhardt.

Another post

Published Dec. 14, 2017, 8:57 a.m. by admin

Post body.

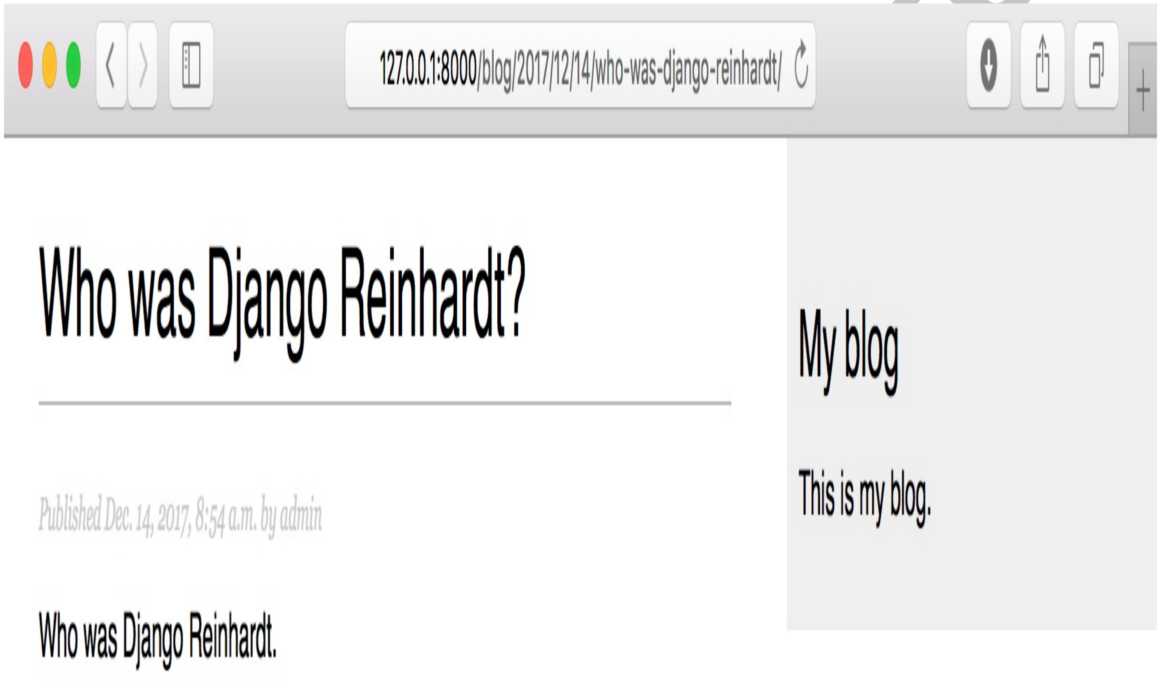
My blog

This is my blog.

Then, let's edit the `post/detail.html` file:

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

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

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



Who was Django Reinhardt?

Published Dec. 14, 2017, 8:54 a.m. by admin

Who was Django Reinhardt.

My blog

This is my blog.

Take a look at the URL—it should be `/blog/2017/12/14/who-was-django-reinhardt/`. We have designed SEO-friendly URLs for our blog posts.

Adding pagination

When you start adding content to your blog, you will soon realize you need to split the list of posts across several pages. Django has a built-in pagination class that allows you to manage paginated data easily.

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

```
from django.core.paginator import Paginator, EmptyPage,\
                                  PageNotAnInteger

def post_list(request):
    object_list = Post.published.all()
    paginator = Paginator(object_list, 3) # 3 posts in each page
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
        # If page is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'page': page,
                   'posts': posts})
```

This is how pagination works:

1. We instantiate the `Paginator` class with the number of objects we want to display on each page.
2. We get the `page GET` parameter that indicates the current page number.

3. We obtain the objects for the desired page calling the `page()` method of `Paginator`.
4. If the `page` parameter is not an integer, we retrieve the first page of results. If this parameter is a number higher than the last page of results, we will retrieve the last page.
5. We pass the page number and retrieved objects to the template.

Now, we have to create a template to display the paginator so that it can be included in any template that uses pagination. In the `templates/` folder of the `blog` application, create a new file and name it `pagination.html`. Add the following HTML code to the file:

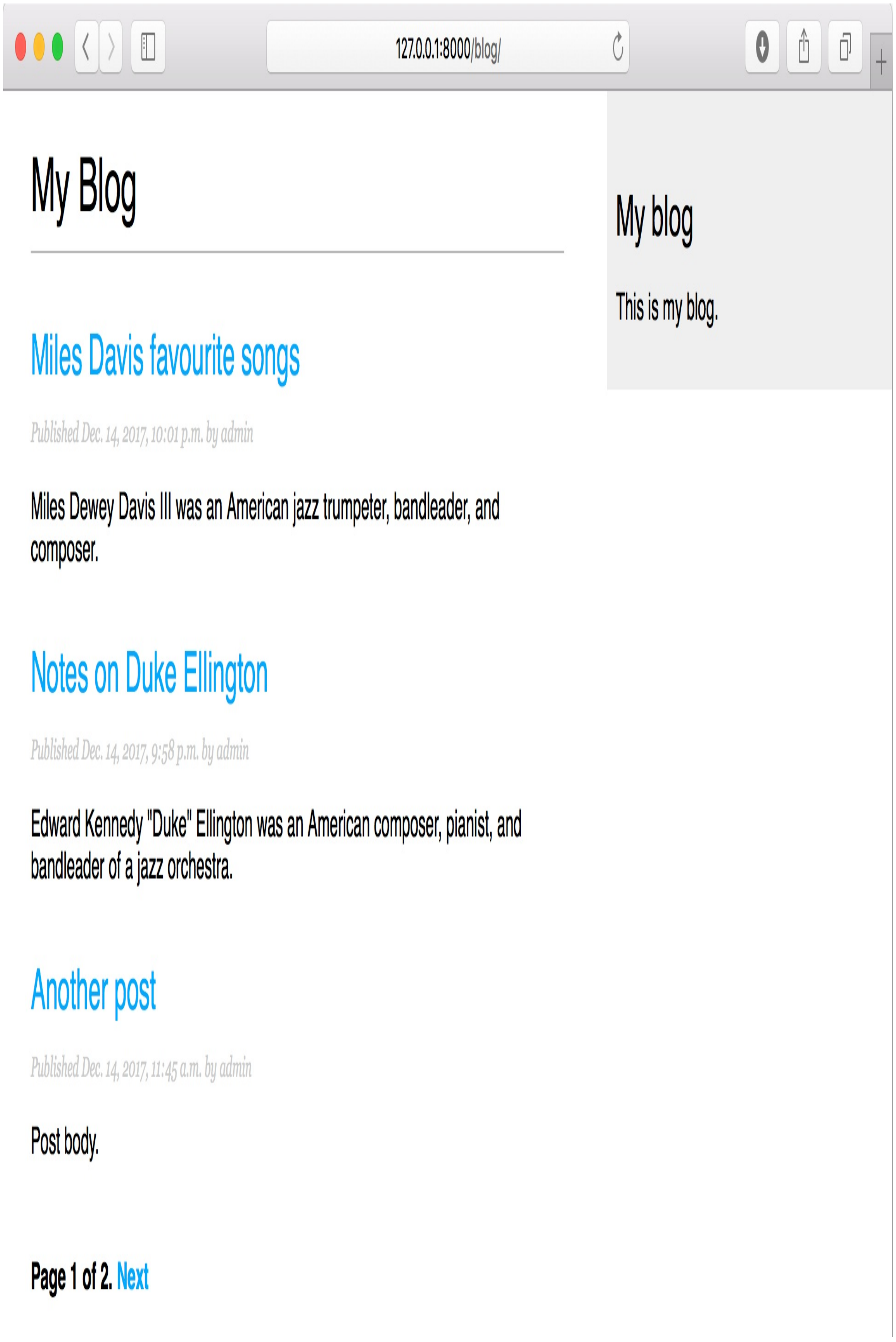
```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
      <a href="?page={{ page.previous_page_number }}">Previous</a>
    {% endif %}
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}.
    </span>
      {% if page.has_next %}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

The pagination template expects a `Page` object in order to render previous and next links and to display the current page and total pages of results. Let's return to the `blog/post/list.html` template and include the `pagination.html` template at the bottom of the `{% content %}` block, as follows:

```
{% block content %}
  ...
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

Since the `Page` object we are passing to the template is called `posts`, we include the pagination template in the post list template, passing the parameters to render it correctly. You can follow this method to reuse your pagination template in paginated views of different models.

Now, open `http://127.0.0.1:8000/blog/` in your browser. You should see the pagination at the bottom of the post list and should be able to navigate through pages:



127.0.0.1:8000/blog/

My Blog

Miles Davis favourite songs

Published Dec. 14, 2017, 10:01 p.m. by admin

Miles Dewey Davis III was an American jazz trumpeter, bandleader, and composer.

Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

Another post

Published Dec. 14, 2017, 11:45 a.m. by admin

Post body.

Page 1 of 2. Next

My blog

This is my blog.

Using class-based views

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

Class-based views offer advantages over function-based views for some use cases. They have the following features:

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

You can take a look at an introduction to class-based views at `https://docs.djangoproject.com/en/2.0/topics/class-based-views/intro/`.

We will change our `post_list` view into a class-based view to use the generic `ListView` offered by Django. This base view allows you to list objects of any kind.

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

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

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

This class-based view is analogous to the previous `post_list` view. In the preceding code, we are telling `ListView` to do the following things:

- Use a specific QuerySet instead of retrieving all objects. Instead of defining a `queryset` attribute, we could have specified `model = Post` and Django would have built the generic `Post.objects.all()` QuerySet for us.
- Use the context variable `posts` for the query results. The default variable is `object_list` if we don't specify any `context_object_name`.
- Paginate the result displaying three objects per page.
- Use a custom template to render the page. If we don't set a default template, `ListView` will use `blog/post_list.html`.

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

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

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

```
{% include "pagination.html" with page=page_obj %}
```

Open `http://127.0.0.1:8000/blog/` in your browser and verify that everything works the same way as with the previous `post_list` view. This is a simple example of a class-based view that uses a generic class provided by Django. You will learn more about class-based views in `Chapter 10`, *Building an E-Learning Platform*, and successive chapters.

Summary

In this chapter, you have learned the basics of the Django web framework by creating a basic blog application. You have designed the data models and applied migrations to your project. You have created the views, templates, and URLs for your blog, including object pagination.

In the next chapter, you will learn how to enhance your blog application with a comment system and tagging functionality and allow your users to share posts by email.

Enhancing Your Blog with Advanced Features

In the preceding chapter, you created a basic blog application. Now, you will turn your application into a fully functional blog with advanced features, such as sharing posts by email, adding comments, tagging posts, and retrieving posts by similarity. In this chapter, you will learn the following topics:

- Sending emails with Django
- Creating forms and handling them in views
- Creating forms from models
- Integrating third-party applications
- Building complex QuerySets

Sharing posts by email

First, we will allow users to share posts by sending them emails. Take a short time to think how you would use *views*, *URLs*, and *templates* to create this functionality using what you have learned in the preceding chapter. Now, check what you need in order to allow your users to send posts by email. You will need to do the following things:

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

Creating forms with Django

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

Django comes with two base classes to build forms:

- `Form`: Allows you to build standard forms
- `ModelForm`: Allows you to build forms tied to model instances

First, create a `forms.py` file inside the directory of your `blog` application and make it look like this:

```
from django import forms

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

This is your first Django form. Take a look at the code. We have created a form by inheriting the base `Form` class. We use different field types for Django to validate fields accordingly.

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

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

Field validation also depends on the field type. For example, the `email` and `to` fields are `EmailField` fields. Both fields require a valid email address, otherwise, the field validation will raise a `forms.ValidationError` exception and the form will not validate. Other parameters are also taken into account for form validation: we define a maximum length of 25 characters for the `name` field and make the `comments` field optional with `required=False`. All of this is also taken into account for field validation. The field types used in this form are only a part of Django form fields. For a list of all form fields available, you can visit `https://docs.djangoproject.com/en/2.0/ref/forms/fields/`.

Handling forms in views

You have to create a new view that handles the form and sends an email when it's successfully submitted. Edit the `views.py` file of your `blog` application and add the following code to it:

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # Retrieve post by id
    post = get_object_or_404(Post, id=post_id, status='published')

    if request.method == 'POST':
        # Form was submitted
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Form fields passed validation
            cd = form.cleaned_data
            # ... send email
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form})
```

This view works as follows:

- We define the `post_share` view that takes the `request` object and the `post_id` variable as parameters.
- We use the `get_object_or_404()` shortcut to retrieve the post by ID and make sure that the retrieved post has a `published` status.
- We use the same view for both displaying the initial form and processing the submitted data. We differentiate whether

the form was submitted or not based on the `request` method and submit the form using `POST`. We assume that if we get a `GET` request, an empty form has to be displayed, and if we get a `POST` request, the form is submitted and needs to be processed. Therefore, we use `request.method == 'POST'` to distinguish between the two scenarios.

The following is the process to display and handle the form:

1. When the view is loaded initially with a `GET` request, we create a new `form` instance that will be used to display the empty form in the template:

```
form = EmailPostForm()
```

2. The user fills in the form and submits it via `POST`. Then, we create a form instance using the submitted data that is contained in `request.POST`:

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

3. After this, we validate the submitted data using the form's `is_valid()` method. This method validates the data introduced in the form and returns `True` if all fields contain valid data. If any field contains invalid data, then `is_valid()` returns `False`. You can see a list of validation errors by accessing `form.errors`.
4. If the form is not valid, we render the form in the template

again with the submitted data. We will display validation errors in the template.

5. If the form is valid, we retrieve the validated data accessing `form.cleaned_data`. This attribute is a dictionary of form fields and their values.

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

Now, let's learn how to send emails using Django to put everything together.

Sending emails with Django

Sending emails with Django is pretty straightforward. First, you will need to have a local SMTP server or define the configuration of an external SMTP server by adding the following settings in the `settings.py` file of your project:

- `EMAIL_HOST`: The SMTP server host; the default is `localhost`
- `EMAIL_PORT`: The SMTP port; the default is `25`
- `EMAIL_HOST_USER`: Username for the SMTP server
- `EMAIL_HOST_PASSWORD`: Password for the SMTP server
- `EMAIL_USE_TLS`: Whether to use a TLS secure connection
- `EMAIL_USE_SSL`: Whether to use an implicit TLS secure connection

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

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

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

If you want to send emails, but you don't have a local SMTP server, you can probably use the SMTP server of your email service provider. The following sample configuration is valid for sending emails via Gmail servers using a Google account:

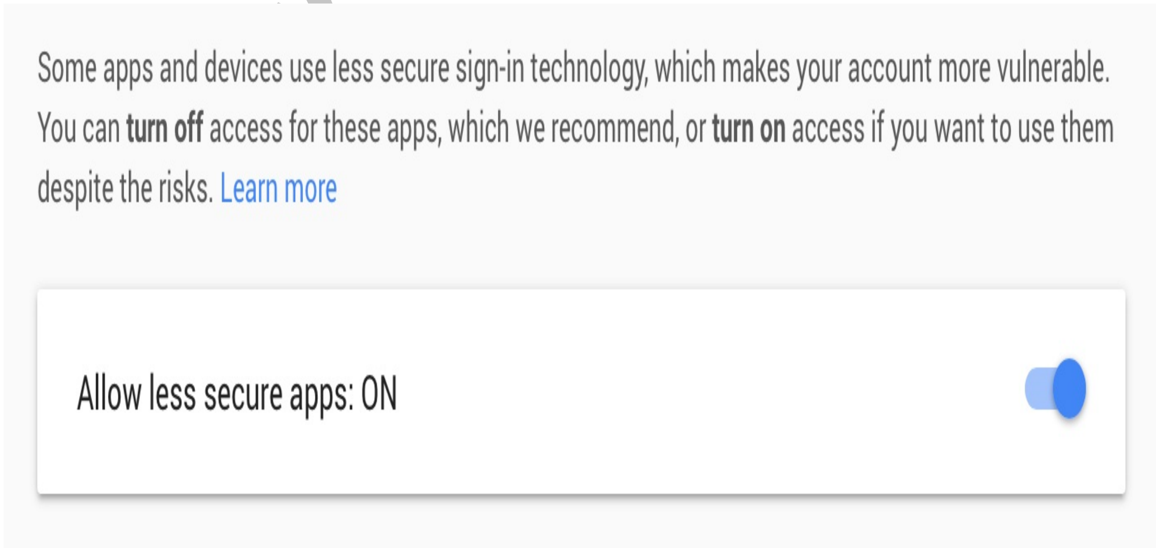
```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'your_password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Run the `python manage.py shell` command to open the Python shell and send an email, as follows:

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

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

If you are sending emails by Gmail with the preceding configuration, you might have to enable access for less secured apps at `https://myaccount.google.com/lesssecureapps`, as follows:



Some apps and devices use less secure sign-in technology, which makes your account more vulnerable. You can **turn off** access for these apps, which we recommend, or **turn on** access if you want to use them despite the risks. Learn more

Allow less secure apps: ON

Now, we will add this functionality to our view.

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

```
from django.core.mail import send_mail

def post_share(request, post_id):
    # Retrieve post by id
    post = get_object_or_404(Post, id=post_id, status='published')
    sent = False

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

We declare a `sent` variable and set it to `True` when the post was sent. We will use that variable later in the template to display a success message when the form is successfully submitted. Since we have to include a link to the post in the email, we will retrieve the absolute path of the post using its `get_absolute_url()` method. We use this path as an input for `request.build_absolute_uri()` to build a complete URL, including HTTP schema and hostname. We build the subject and the message body of the email using the cleaned data of the validated form and finally send the email to the email address contained in the `to` field of the form.

Now that your view is complete, remember to add a new URL pattern for it. Open the `urls.py` file of your `blog` application and add the `post_share` URL pattern, as follows:

```
urlpatterns = [
    # ...
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
]
```

Rendering forms in templates

After creating the form, programming the view, and adding the URL pattern, we are only missing the template for this view. Create a new file in the `blog/templates/blog/post/` directory and name it `share.html`; add the following code to it:

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

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

{% block content %}
  {% if sent %}
    <h1>E-mail successfully sent</h1>
    <p>
      "{{ post.title }}" was successfully sent to {{ form.cleaned_data.to }}.
    </p>
  {% else %}
    <h1>Share "{{ post.title }}" by e-mail</h1>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="submit" value="Send e-mail">
    </form>
  {% endif %}
{% endblock %}
```

This is the template to display the form or a success message when it's sent. As you would notice, we create the HTML form element, indicating that it has to be submitted by the `POST` method:

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

Then, we include the actual form instance. We tell Django to render its fields in HTML paragraph `<p>` elements with the `as_p` method. We can also render the form as an unordered list with `as_ul` or as an HTML table with `as_table`. If we want to render each field, we can also iterate through the fields, as in the following example:

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

The `{% csrf_token %}` template tag introduces a hidden field with an autogenerated token to avoid **cross-site request forgery** (**CSRF**) attacks. These attacks consist of a malicious website or program performing an unwanted action for a user on your site. You can find more information about this at `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`.

The preceding tag generates a hidden field that looks like this:

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

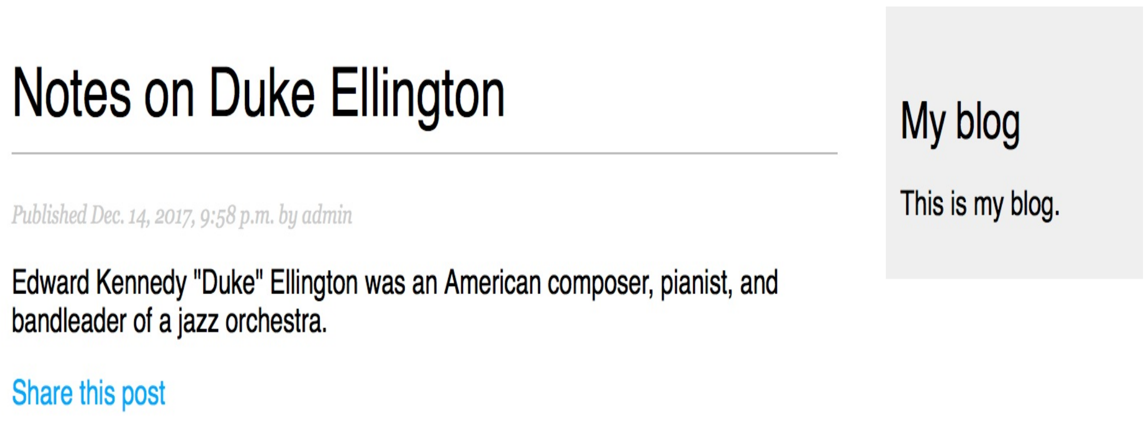
By default, Django checks for the CSRF token in all `POST` requests. Remember that you include the `csrf_token` tag in all forms that are submitted via `POST`.

Edit your `blog/post/detail.html` template and add the following link to the share post URL after the `{{ post.body|linebreaks }}` variable:

```
<p>
  <a href="{% url "blog:post_share" post.id %}">
    Share this post
  </a>
</p>
```

Remember that we are building the URL dynamically using the `{% url %}` template tag provided by Django. We are using the namespace called `blog` and the URL named `post_share`, and we are passing the post ID as a parameter to build the absolute URL.

Now, start the development server with the `python manage.py` `runserver` command and open `http://127.0.0.1:8000/blog/` in your browser. Click on any post title to view its detail page. Under the post body, you should see the link we just added, as shown in the following screenshot:



Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

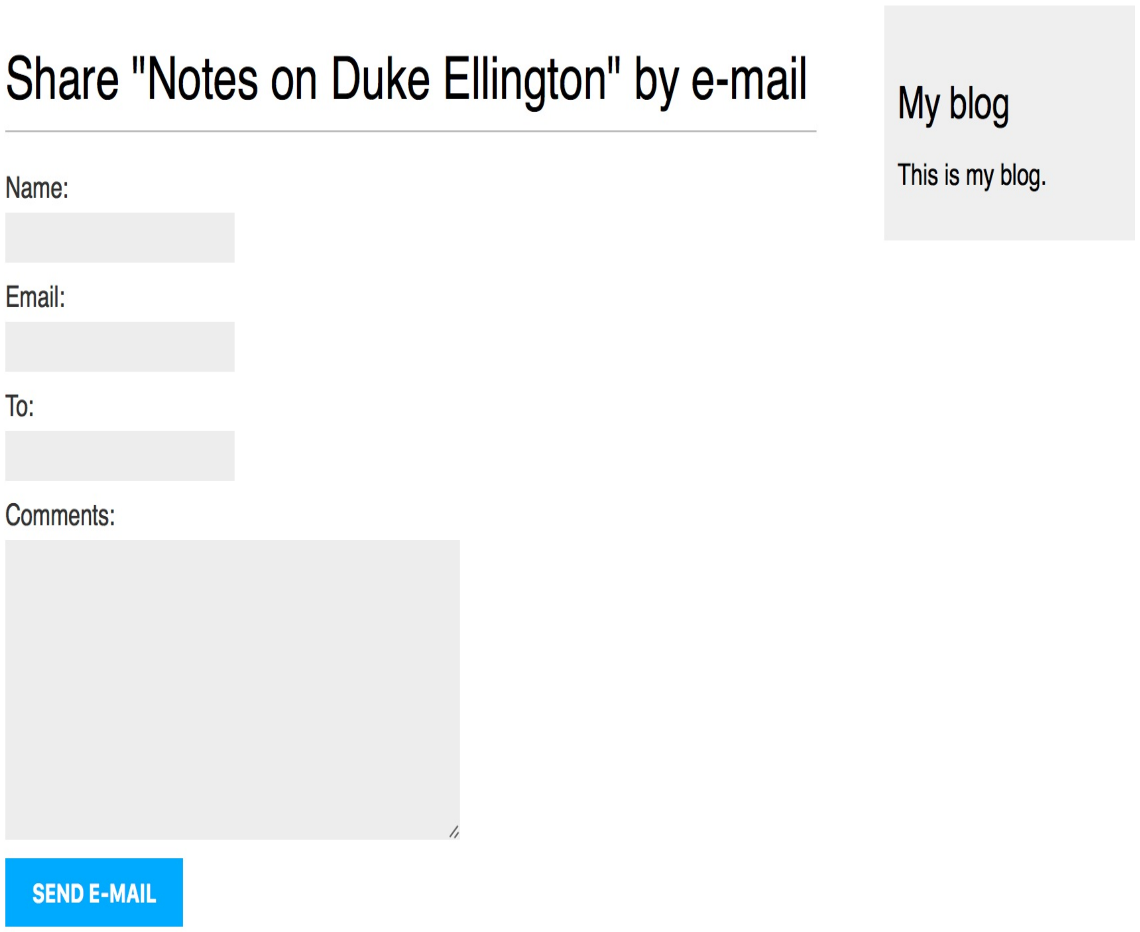
Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

Share this post

My blog

This is my blog.

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


Share "Notes on Duke Ellington" by e-mail

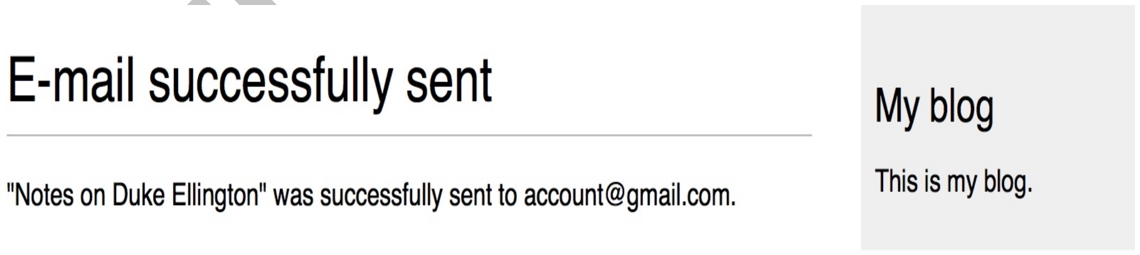
Name:

Email:

To:

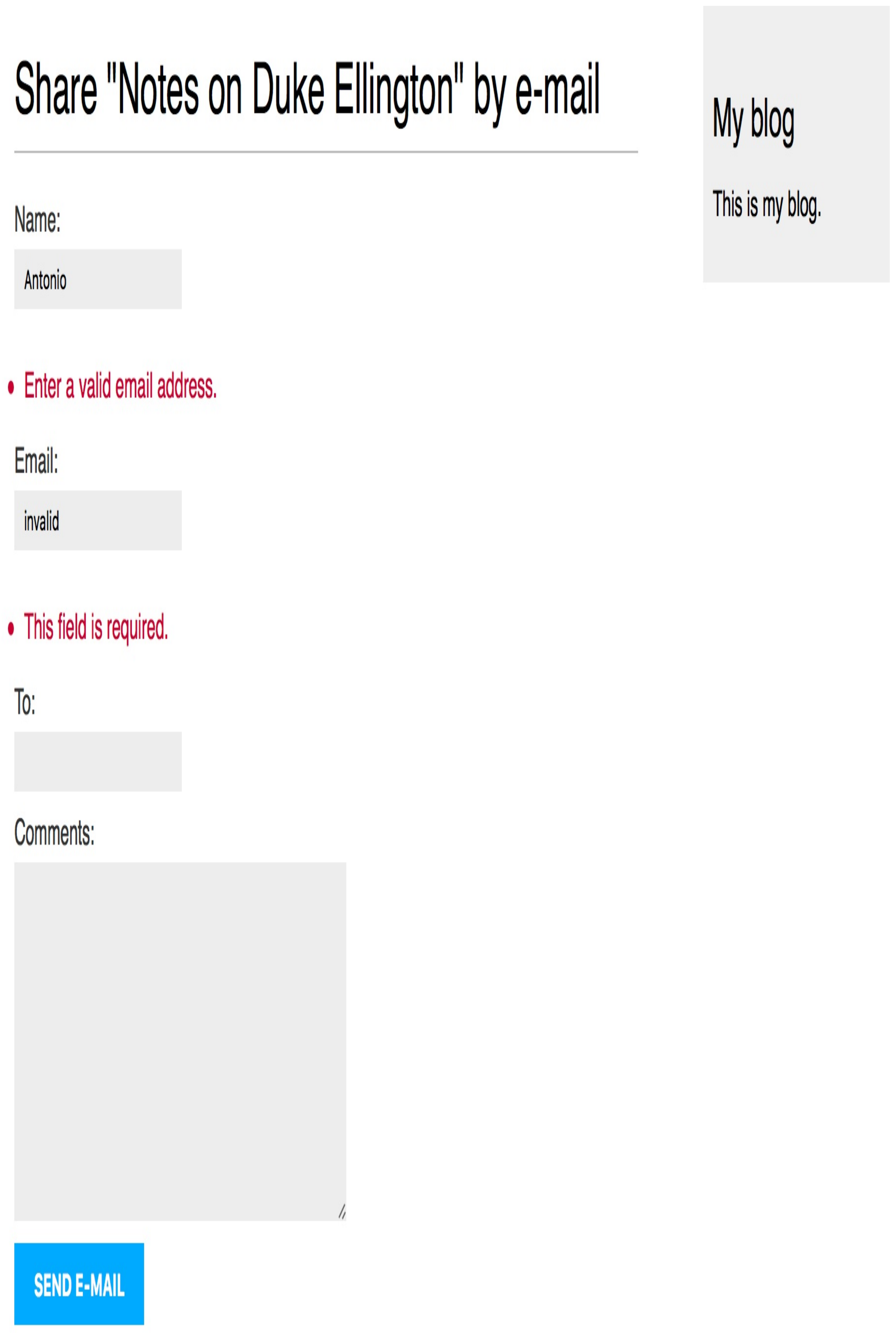
Comments:

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


E-mail successfully sent

"Notes on Duke Ellington" was successfully sent to account@gmail.com.

If you input invalid data, you will see that the form is rendered again, including all validation errors:



Share "Notes on Duke Ellington" by e-mail

Name:

Antonio

- Enter a valid email address.

Email:

invalid

- This field is required.

To:

Comments:

My blog

This is my blog.

Note that some modern browsers will prevent you from submitting the form with empty or erroneous fields. This is because of form validation done by the browser based on field types and restrictions per field. In this case, the form won't be submitted and the browser will display an error message for the fields that are wrong.

Our form for sharing posts by email is now complete. Let's create a comment system for our blog.

Creating a comment system

Now, we will build a comment system for the blog, wherein the users will be able to comment on posts. To build the comment system, you will need to do the following steps:

1. Create a model to save comments
2. Create a form to submit comments and validate the input data
3. Add a view that processes the form and saves the new comment to the database
4. Edit the post detail template to display the list of comments and the form to add a new comment

First, let's build a model to store comments. Open the `models.py` file of your `blog` application and add the following code:

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

    class Meta:
        ordering = ('created',)

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

This is our `Comment` model. It contains `ForeignKey` to associate the comment with a single post. This many-to-one relationship is defined in the `Comment` model because each comment will be made on one post, and each post may have multiple comments. The `related_name` attribute allows us to name the attribute that we use for the relation from the related object back to this one. After defining this, we can retrieve the post of a comment object using `comment.post` and retrieve all comments of a post using `post.comments.all()`. If you don't define the `related_name` attribute, Django will use the name of the model in lowercase, followed by `_set` (that is, `comment_set`) to name the manager of the related object back to this one.

You can learn more about many-to-one relationships at https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_one/.

We have included an `active` boolean field that we will use to manually deactivate inappropriate comments. We use the `created` field to sort comments in a chronological order by default.

The new `Comment` model you just created is not yet synchronized into the database. Run the following command to generate a new migration that reflects the creation of the new model:

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

You should see the following output:

```
Migrations for 'blog':
  blog/migrations/0002_comment.py
    - Create model Comment
```

Django has generated a `0002_comment.py` file inside the `migrations/` directory of the `blog` application. Now, you will need to create the related database schema and apply the changes to the database. Run the following command to apply existing migrations:

```
python manage.py migrate
```

You will get an output that includes the following line:

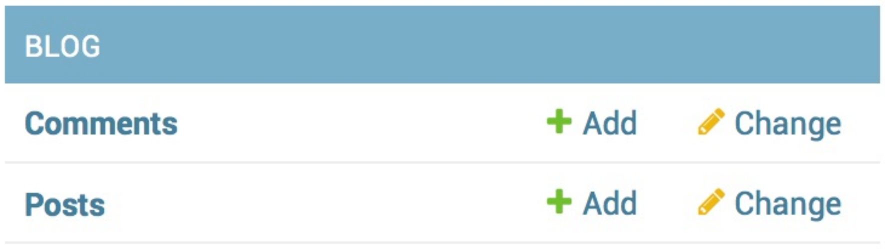
```
Applying blog.0002_comment... OK
```

The migration we just created has been applied, and, now, a `blog_comment` table exists in the database.

Now, we can add our new model to the administration site in order to manage comments through a simple interface. Open the `admin.py` file of the `blog` application, import the `Comment` model, and add the following `ModelAdmin` class:

```
from .models import Post, Comment

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

Start the development server with the `python manage.py runserver` command and open `http://127.0.0.1:8000/admin/` in your browser. You should see the new model included in the BLOG section, as shown in the following screenshot:



The model is now registered in the admin site, and we can manage `Comment` instances using a simple interface.

Creating forms from models

We will still need to build a form to let our users comment on blog posts. Remember that Django has two base classes to build forms, `Form` and `ModelForm`. You used the first one previously to let your users share posts by email. In the present case, you will need to use `ModelForm` because you have to build a form dynamically from your `Comment` model. Edit the `forms.py` file of your `blog` application and add the following lines:

```
from .models import Comment

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

To create a form from a model, we will just need to indicate which model to use to build the form in the `Meta` class of the form. Django introspects the model and builds the form dynamically for us. Each model field type has a corresponding default form field type. The way we define our model fields is taken into account for form validation. By default, Django builds a form field for each field contained in the model. However, you can explicitly tell the framework which fields you want to include in your form using a `fields` list or define which fields you want to exclude using an `exclude` list of fields. For our `CommentForm` form, we will just use the `name`, `email`, and `body` fields because those are the only fields our users will be able to fill in.

Handling ModelForms in views

We will use the post detail view to instantiate the form and process it in order to keep it simple. Edit the `views.py` file, add imports for the `Comment` model and the `CommentForm` form, and modify the `post_detail` view to make it look like the following:

```
from .models import Post, Comment
from .forms import EmailPostForm, CommentForm

def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post,
                                   status='published',
                                   publish__year=year,
                                   publish__month=month,
                                   publish__day=day)

    # List of active comments for this post
    comments = post.comments.filter(active=True)

    new_comment = None

    if request.method == 'POST':
        # A comment was posted
        comment_form = CommentForm(data=request.POST)
        if comment_form.is_valid():
            # Create Comment object but don't save to database yet
            new_comment = comment_form.save(commit=False)
            # Assign the current post to the comment
            new_comment.post = post
            # Save the comment to the database
            new_comment.save()
    else:
        comment_form = CommentForm()
    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                   'comments': comments,
                   'new_comment': new_comment,
                   'comment_form': comment_form})
```

Let's review what we have added to our view. We used the `post_detail` view to display the post and its comments. We added a QuerySet to retrieve all active comments for this post, as follows:

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

We build this QuerySet, starting from the `post` object. We use the manager for related objects we defined as `comments` using the `related_name` attribute of the relationship in the `Comment` model.

We also use the same view to let our users add a new comment. Therefore, we initialize the `new_comment` variable by setting it to `None`. We will use this variable when a new comment is created. We build a form instance with `comment_form = CommentForm()` if the view is called by a `GET` request. If the request is done via `POST`, we instantiate the form using the submitted data and validate it using the `is_valid()` method. If the form is invalid, we render the template with the validation errors. If the form is valid, we take the following actions:

1. We create a new `Comment` object by calling the form's `save()` method and assign it to the `new_comment` variable as follows:

```
    new_comment = comment_form.save(commit=False)
```

   The `save()` method creates an instance of the model that the form is linked to and saves it to the database. If you call it using `commit=False`, you create the model instance, but you don't save it to the database yet. This comes in handy when you want to modify the object before finally saving it, which is what we do next.

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

2. We assign the current post to the comment we just created:

```
new_comment.post = post
```

   By doing this, we are specifying that the new comment belongs to this post.

3. Finally, we save the new comment to the database by calling its `save()` method:

```
new_comment.save()
```

Our view is now ready to display and process new comments.

Adding comments to the post detail template

We have created the functionality to manage comments for a post. Now, we will need to adapt our `post/detail.html` template to do the following things:

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

First, we will add the total comments. Open the `post/detail.html` template and append the following code to the `content` block:

```
{% with comments.count as total_comments %}
  <h2>
    {{ total_comments }} comment{{ total_comments|pluralize }}
  </h2>
{% endwith %}
```

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

> *The `{% with %}` template tag is useful to avoid hitting the database or accessing expensive methods multiple times.*

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

The `pluralize` template filter returns a string with the letter *"s"* if the value is different from `1`. The preceding text will be rendered as *0 comments*, *1 comment*, or *N comments*. Django includes plenty of template tags and filters that help you display information in the way you want.

Now, let's include the list of comments. Append the following lines to the `post/detail.html` template below the preceding code:

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

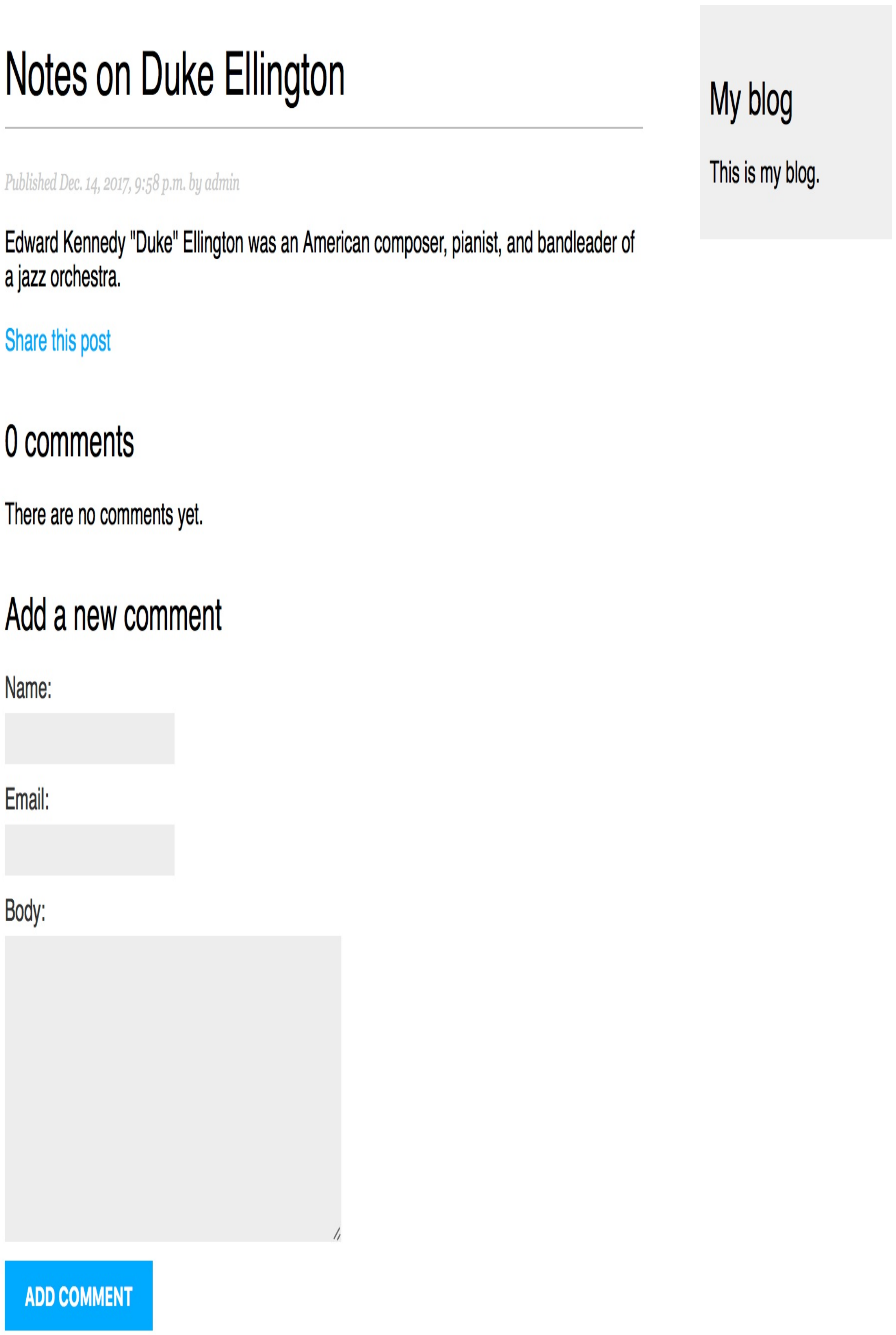
We use the `{% for %}` template tag to loop through comments. We display a default message if the `comments` list is empty, informing our users that there are no comments on this post yet. We enumerate comments with the `{{ forloop.counter }}` variable, which contains the loop counter in each iteration. Then, we display the name of the user who posted the comment, the date, and the body of the comment.

Finally, you need to render the form or display a successful message instead when it is successfully submitted. Add the following lines just below the preceding code:

```
{% if new_comment %}
  <h2>Your comment has been added.</h2>
```

```
{% else %}
  <h2>Add a new comment</h2>
  <form action="." method="post">
    {{ comment_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Add comment"></p>
  </form>
{% endif %}
```

The code is pretty straightforward: if the `new_comment` object exists, we display a success message because the comment was successfully created. Otherwise, we render the form with a paragraph `<p>` element for each field and include the CSRF token required for `POST` requests. Open `http://127.0.0.1:8000/blog/` in your browser and click on a post title to take a look at its detail page. You will see something like the following screenshot:



Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

Share this post

0 comments

There are no comments yet.

Add a new comment

Name:

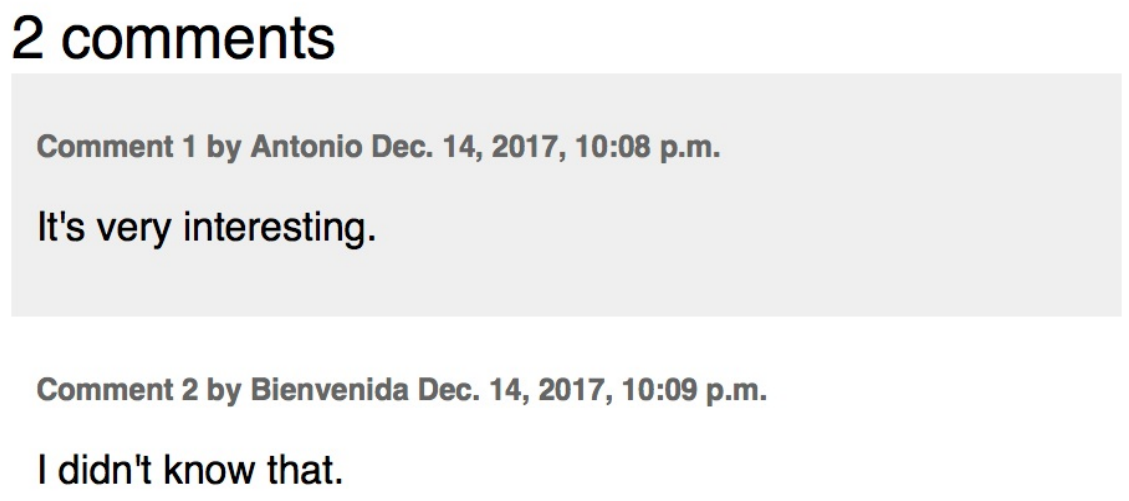
Email:

Body:

My blog

This is my blog.

Add a couple of comments using the form. They should appear under your post in chronological order, as follows:


2 comments

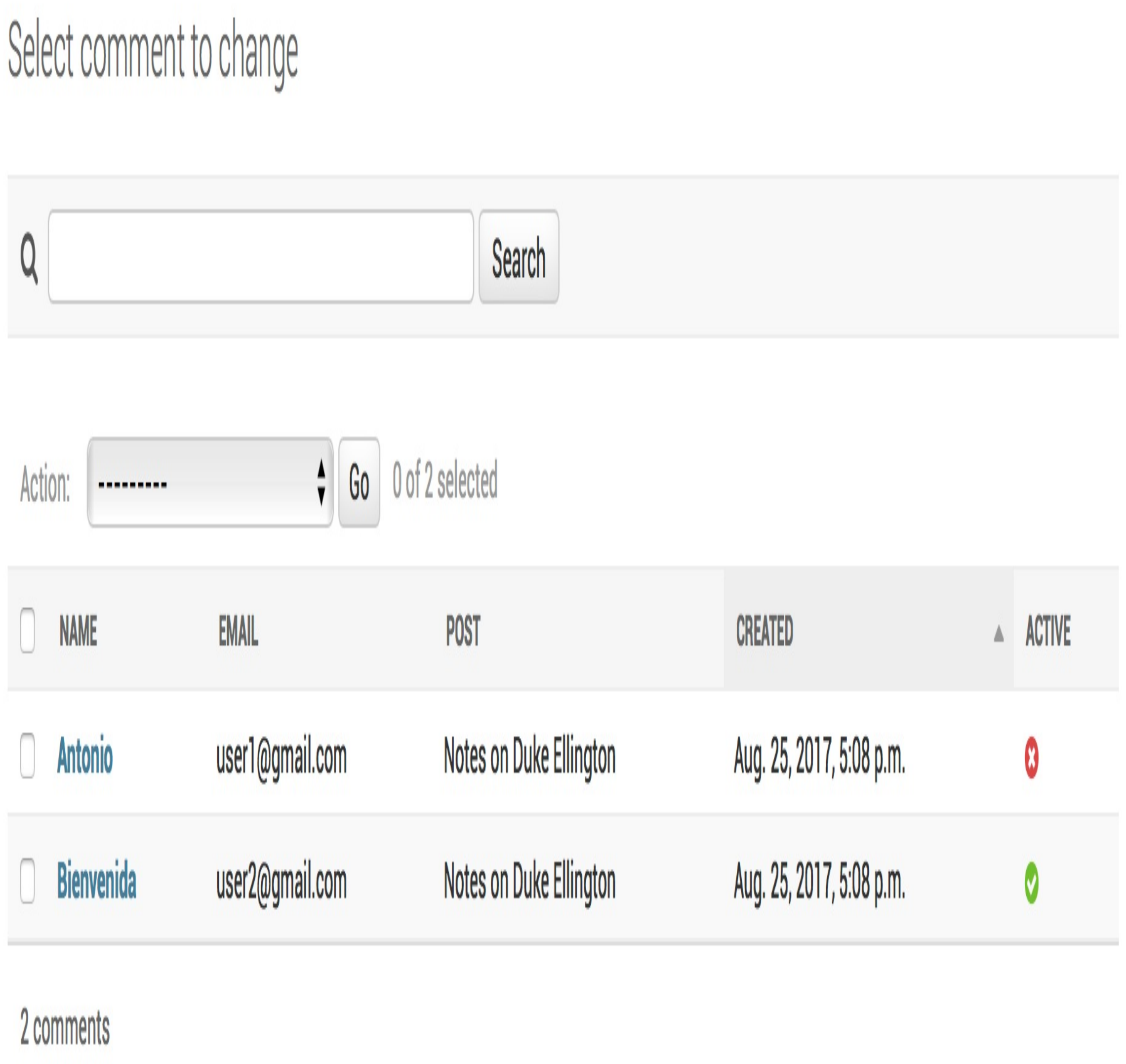
Comment 1 by Antonio Dec. 14, 2017, 10:08 p.m.

It's very interesting.

Comment 2 by Bienvenida Dec. 14, 2017, 10:09 p.m.

I didn't know that.

Open `http://127.0.0.1:8000/admin/blog/comment/` in your browser. You will see the admin page with the list of comments you created. Click on one of them to edit it, uncheck the Active checkbox, and click on the Save button. You will be redirected to the list of comments again, and the Active column will display an inactive icon for the comment. It should look like the first comment in the following screenshot:


Select comment to change

Search

Action: --------- Go 0 of 2 selected

	NAME	EMAIL	POST	CREATED	ACTIVE
☐	Antonio	user1@gmail.com	Notes on Duke Ellington	Aug. 25, 2017, 5:08 p.m.	✗
☐	Bienvenida	user2@gmail.com	Notes on Duke Ellington	Aug. 25, 2017, 5:08 p.m.	✓

2 comments

If you return to the post detail view, you will note that the deleted comment is not displayed any more; neither is it being counted for the total number of comments. Thanks to the `active` field, you can deactivate inappropriate comments and avoid showing them in your posts.