Fill in the form with your current password and your new password and click on the **CHANGE** button. You will see the following success page:

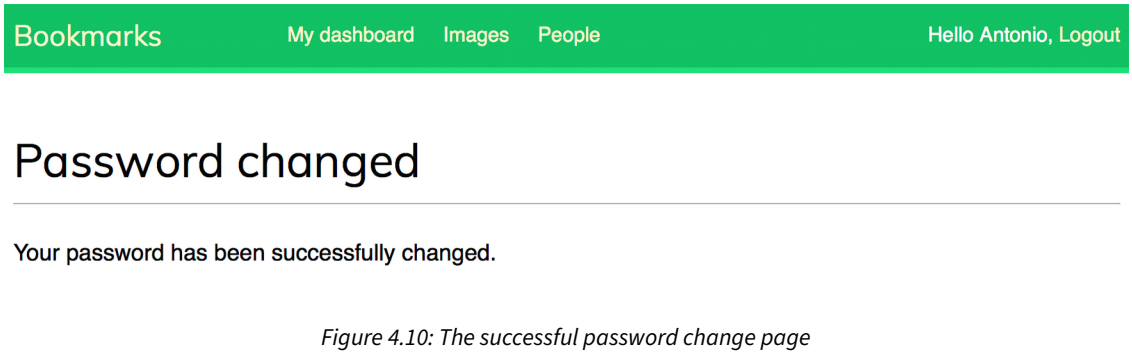


Figure 4.10: The successful password change page

Log out and log in again using your new password to verify that everything works as expected.

Reset password views

Edit the `urls.py` file of the `account` application and add the following URL patterns highlighted in bold:

```
urlpatterns = [
    # previous login url
    # path('login/', views.user_login, name='login'),

    # login / logout urls
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # change password urls
    path('password-change/',
         auth_views.PasswordChangeView.as_view(),
         name='password_change'),
    path('password-change/done/',
          auth_views.PasswordChangeDoneView.as_view(),
          name='password_change_done'),

    # reset password urls
    path('password-reset/',
         auth_views.PasswordResetView.as_view(),
         name='password_reset'),
    path('password-reset/done/',
         auth_views.PasswordResetDoneView.as_view(),
         name='password_reset_done'),
    path('password-reset/<uidb64>/<token>/',
```

```
        auth_views.PasswordResetConfirmView.as_view(),
        name='password_reset_confirm'),
    path('password-reset/complete/',
        auth_views.PasswordResetCompleteView.as_view(),
        name='password_reset_complete'),

    path('', views.dashboard, name='dashboard'),
]
```

Add a new file in the `templates/registration/` directory of the `account` application and name it `password_reset_form.html`. Add the following code to it:

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

{% block title %}Reset your password{% endblock %}

{% block content %}
  <h1>Forgotten your password?</h1>
  <p>Enter your e-mail address to obtain a new password.</p>
  <form method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Send e-mail"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

Now create another file in the same directory and name it `password_reset_email.html`. Add the following code to it:

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol }}://{{ domain }}{% url "password_reset_confirm" uidb64=uid
token=token %}
Your username, in case you've forgotten: {{ user.get_username }}
```

The `password_reset_email.html` template will be used to render the email sent to users to reset their password. It includes a reset token that is generated by the view.

Create another file in the same directory and name it `password_reset_done.html`. Add the following code to it:

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

{% block title %}Reset your password{% endblock %}

{% block content %}
```

```
  <h1>Reset your password</h1>
  <p>We've emailed you instructions for setting your password.</p>
  <p>If you don't receive an email, please make sure you've entered the address
you registered with.</p>
{% endblock %}
```

Create another template in the same directory and name it `password_reset_confirm.html`. Add the following code to it:

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

{% block title %}Reset your password{% endblock %}

{% block content %}
  <h1>Reset your password</h1>
  {% if validlink %}
    <p>Please enter your new password twice:</p>
    <form method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Change my password" /></p>
    </form>
  {% else %}
    <p>The password reset link was invalid, possibly because it has already
been used. Please request a new password reset.</p>
  {% endif %}
{% endblock %}
```

In this template, we confirm whether the link for resetting the password is valid by checking the `validlink` variable. The view `PasswordResetConfirmView` checks the validity of the token provided in the URL and passes the `validlink` variable to the template. If the link is valid, the user password reset form is displayed. Users can only set a new password if they have a valid reset password link.

Create another template and name it `password_reset_complete.html`. Enter the following code into it:

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

{% block title %}Password reset{% endblock %}

{% block content %}
  <h1>Password set</h1>
  <p>Your password has been set. You can <a href="{% url "login" %}">log in
now</a></p>
{% endblock %}
```

Finally, edit the `registration/login.html` template of the `account` application, and add the following lines highlighted in bold:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  {% if form.errors %}
    <p>
      Your username and password didn't match.
      Please try again.
    </p>
  {% else %}
    <p>Please, use the following form to log-in:</p>
  {% endif %}
  <div class="login-form">
    <form action="{% url 'login' %}" method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="hidden" name="next" value="{{ next }}" />
      <p><input type="submit" value="Log-in"></p>
    </form>
    <p>
      <a href="{% url "password_reset" %}">
        Forgotten your password?
      </a>
    </p>
  </div>
{% endblock %}
```

Now, open `http://127.0.0.1:8000/account/login/` in your browser. The **Log-in page** should now include a link to the reset password page, as follows:

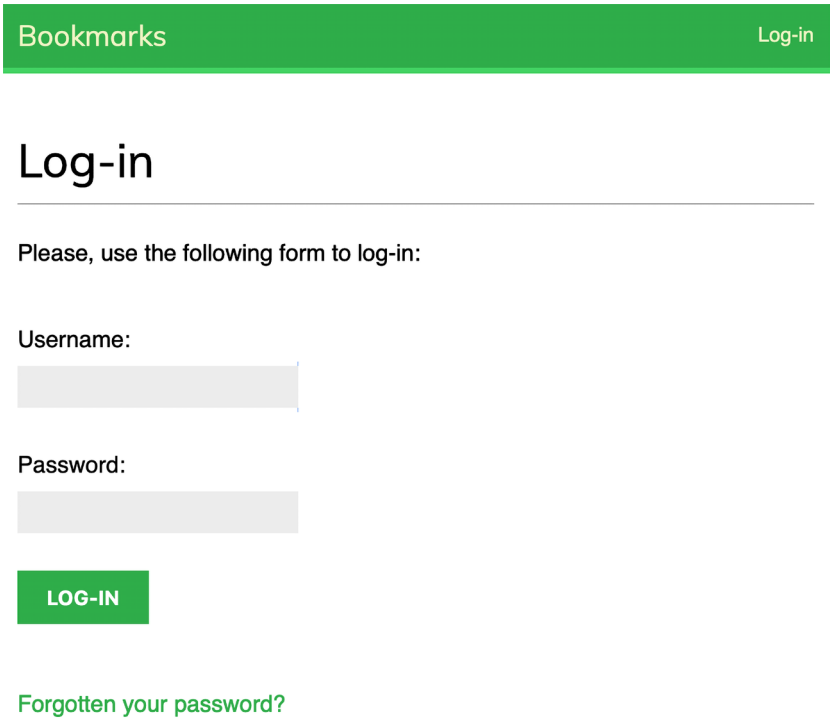


Figure 4.11: The Log-in page including a link to the reset password page

Click on the **Forgotten your password?** link. You should see the following page:

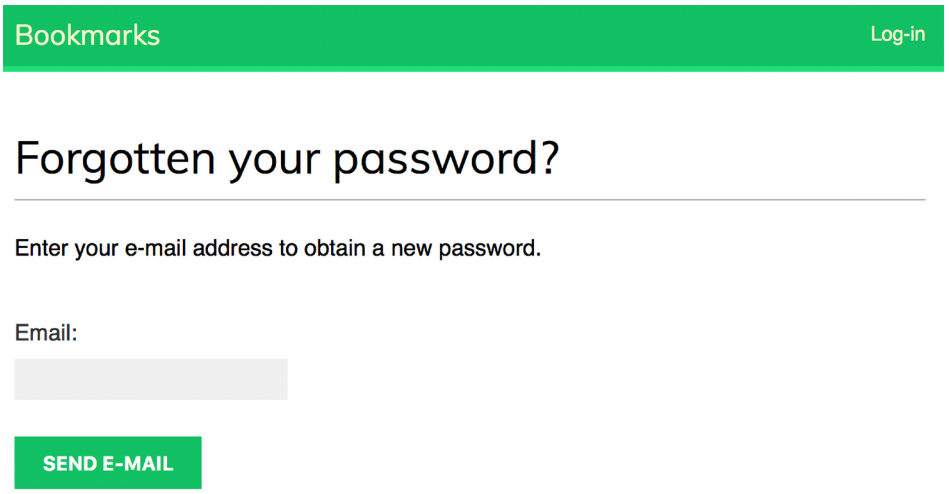


Figure 4.12: The restore password form

At this point, we need to add a **Simple Mail Transfer Protocol** (**SMTP**) configuration to the `settings.py` file of your project so that Django is able to send emails. You learned how to add email settings to your project in *Chapter 2*, *Enhancing Your Blog with Advanced Features*. However, during development, you can configure Django to write emails to the standard output instead of sending them through an SMTP server. Django provides an email backend to write emails to the console.

Edit the `settings.py` file of your project, and add the following line to it:

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

The `EMAIL_BACKEND` setting indicates the class that will be used to send emails.

Return to your browser, enter the email address of an existing user, and click on the **SEND E-MAIL** button. You should see the following page:

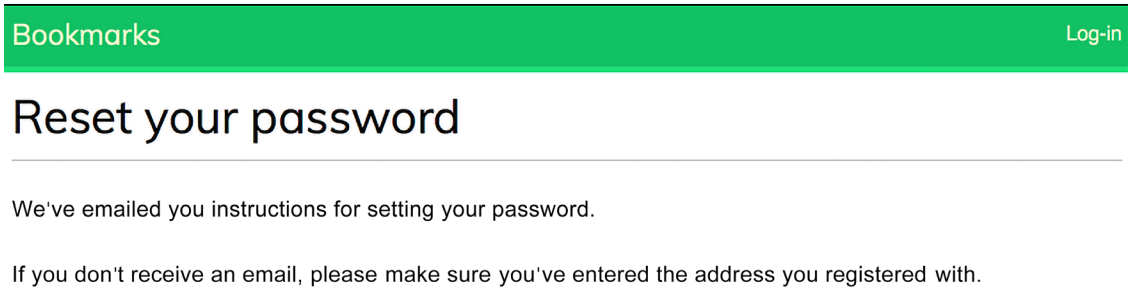


Figure 4.13: The reset password email sent page

Take a look at the shell prompt, where you are running the development server. You will see the generated email, as follows:

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: test@gmail.com
Date: Mon, 10 Jan 2022 19:05:18 -0000
Message-ID: <162896791878.58862.14771487060402279558@MBP-amele.local>

Someone asked for password reset for email test@gmail.com. Follow the link
below:
http://127.0.0.1:8000/account/password-reset/MQ/ardx0u-
b4973cfa2c70d652a190e79054bc479a/
Your username, in case you've forgotten: test
```

The email is rendered using the `password_reset_email.html` template that you created earlier. The URL to reset the password includes a token that was generated dynamically by Django.

Copy the URL from the email, which should look similar to `http://127.0.0.1:8000/account/password-reset/MQ/ardx0u-b4973cfa2c70d652a190e79054bc479a/`, and open it in your browser. You should see the following page:

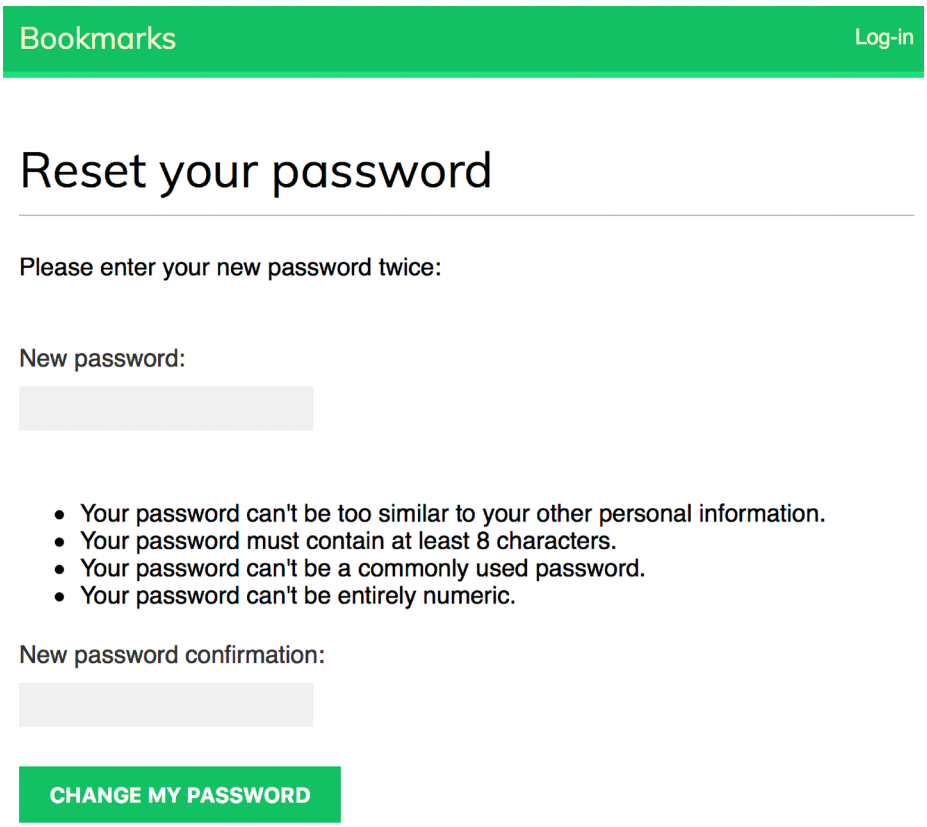


Figure 4.14: The reset password form

The page to set a new password uses the `password_reset_confirm.html` template. Fill in a new password and click on the **CHANGE MY PASSWORD** button. Django will create a new hashed password and save it into the database. You will see the following success page:

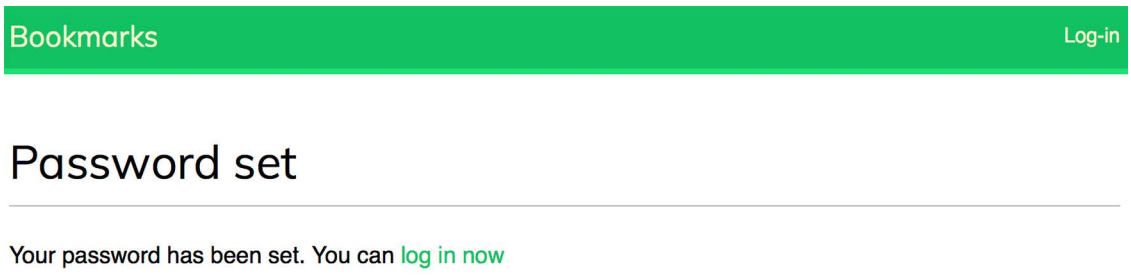


Figure 4.15: The successful password reset page

Now you can log back into the user account using the new password.

Each token to set a new password can be used only once. If you open the link you received again, you will get a message stating that the token is invalid.

We have now integrated the views of the Django authentication framework into the project. These views are suitable for most cases. However, you can create your own views if you need different behavior.

Django provides URL patterns for the authentication views that are equivalent to the ones we just created. We will replace the authentication URL patterns with the ones provided by Django.

Comment out the authentication URL patterns that you added to the `urls.py` file of the `account` application and include `django.contrib.auth.urls` instead, as follows. New code is highlighted in bold:

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

urlpatterns = [
    # previous login view
    # path('login/', views.user_login, name='login'),

    # path('login/', auth_views.LoginView.as_view(), name='login'),
    # path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # change password urls
    # path('password-change/',
    #      auth_views.PasswordChangeView.as_view(),
    #      name='password_change'),
    # path('password-change/done/',
    #       auth_views.PasswordChangeDoneView.as_view(),
    #       name='password_change_done'),

    # reset password urls
    # path('password-reset/',
    #      auth_views.PasswordResetView.as_view(),
    #      name='password_reset'),
    # path('password-reset/done/',
    #      auth_views.PasswordResetDoneView.as_view(),
    #      name='password_reset_done'),
    # path('password-reset/<uidb64>/<token>/',
    #      auth_views.PasswordResetConfirmView.as_view(),
    #      name='password_reset_confirm'),
    # path('password-reset/complete/',
```

```
    #       auth_views.PasswordResetCompleteView.as_view(),
    #       name='password_reset_complete'),

    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
]
```

You can see the authentication URL patterns included at `https://github.com/django/django/blob/stable/4.0.x/django/contrib/auth/urls.py`.

We have now added all the necessary authentication views to our project. Next, we will implement user registration.

User registration and user profiles

Site users can now log in, log out, change their password, and reset their password. However, we need to build a view to allow visitors to create a user account.

User registration

Let's create a simple view to allow user registration on your website. Initially, you have to create a form to let the user enter a username, their real name, and a password.

Edit the `forms.py` file located inside the `account` application directory and add the following lines highlighted in bold:

```
from django import forms
from django.contrib.auth.models import User


class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)


class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'first_name', 'email']
```

We have created a model form for the user model. This form includes the fields `username`, `first_name`, and `email` of the `User` model. These fields will be validated according to the validations of their corresponding model fields. For example, if the user chooses a username that already exists, they will get a validation error because `username` is a field defined with `unique=True`.

We have added two additional fields—`password` and `password2`—for users to set a password and to repeat it. Let's add the field validation to check both passwords are the same.

Edit the `forms.py` file in the `account` application and add the following `clean_password2()` method to the `UserRegistrationForm` class. New code is highlighted in bold:

```
class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'first_name', 'email']

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']
```

We have defined a `clean_password2()` method to compare the second password against the first one and raise a validation error if the passwords don't match. This method is executed when the form is validated by calling its `is_valid()` method. You can provide a `clean_<fieldname>()` method to any of your form fields in order to clean the value or raise form validation errors for a specific field. Forms also include a general `clean()` method to validate the entire form, which is useful to validate fields that depend on each other. In this case, we use the field-specific `clean_password2()` validation instead of overriding the `clean()` method of the form. This avoids overriding other field-specific checks that the `ModelForm` gets from the restrictions set in the model (for example, validating that the `username` is unique).

Django also provides a `UserCreationForm` form that resides in `django.contrib.auth.forms` and is very similar to the one we have created.

Edit the `views.py` file of the `account` application and add the following code highlighted in bold:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
```

```
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm

# ...

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Create a new user object but avoid saving it yet
            new_user = user_form.save(commit=False)
            # Set the chosen password
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Save the User object
            new_user.save()
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

The view for creating user accounts is quite simple. For security reasons, instead of saving the raw password entered by the user, we use the `set_password()` method of the `User` model. This method handles password hashing before storing the password in the database.

Django doesn't store clear text passwords; it stores hashed passwords instead. Hashing is the process of transforming a given key into another value. A hash function is used to generate a fixed-length value according to a mathematical algorithm. By hashing passwords with secure algorithms, Django ensures that user passwords stored in the database require massive amounts of computing time to break.

By default, Django uses the `PBKDF2` hashing algorithm with a SHA256 hash to store all passwords. However, Django not only supports checking existing passwords hashed with `PBKDF2`, but also supports checking stored passwords hashed with other algorithms such as `PBKDF2SHA1`, `argon2`, `bcrypt`, and `scrypt`.

The `PASSWORD_HASHERS` setting defines the password hashers that the Django project supports. The following is the default `PASSWORD_HASHERS` list:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.ScryptPasswordHasher',
]
```

Django uses the first entry of the list, in this case `PBKDF2PasswordHasher`, to hash all passwords. The rest of the hashers can be used by Django to check existing passwords.

The `scrypt` hasher has been introduced in Django 4.0. It is more secure and recommended over `PBKDF2`. However, `PBKDF2` is still the default hasher, as `scrypt` requires OpenSSL 1.1+ and more memory.

You can learn more about how Django stores passwords and about the password hashers included at `https://docs.djangoproject.com/en/4.1/topics/auth/passwords/`.

Now, edit the `urls.py` file of the `account` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [

    # ...

    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
]
```

Finally, create a new template in the `templates/account/` template directory of the `account` application, name it `register.html`, and make it look as follows:

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

{% block title %}Create an account{% endblock %}

{% block content %}
  <h1>Create an account</h1>
  <p>Please, sign up using the following form:</p>
  <form method="post">
    {{ user_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Create my account"></p>
  </form>
{% endblock %}
```

Create an additional template file in the same directory and name it `register_done.html`. Add the following code to it:

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

{% block title %}Welcome{% endblock %}

{% block content %}
  <h1>Welcome {{ new_user.first_name }}!</h1>
  <p>
    Your account has been successfully created.
    Now you can <a href="{% url "login" %}">log in</a>.
  </p>
{% endblock %}
```

Open `http://127.0.0.1:8000/account/register/` in your browser. You will see the registration page you have created:

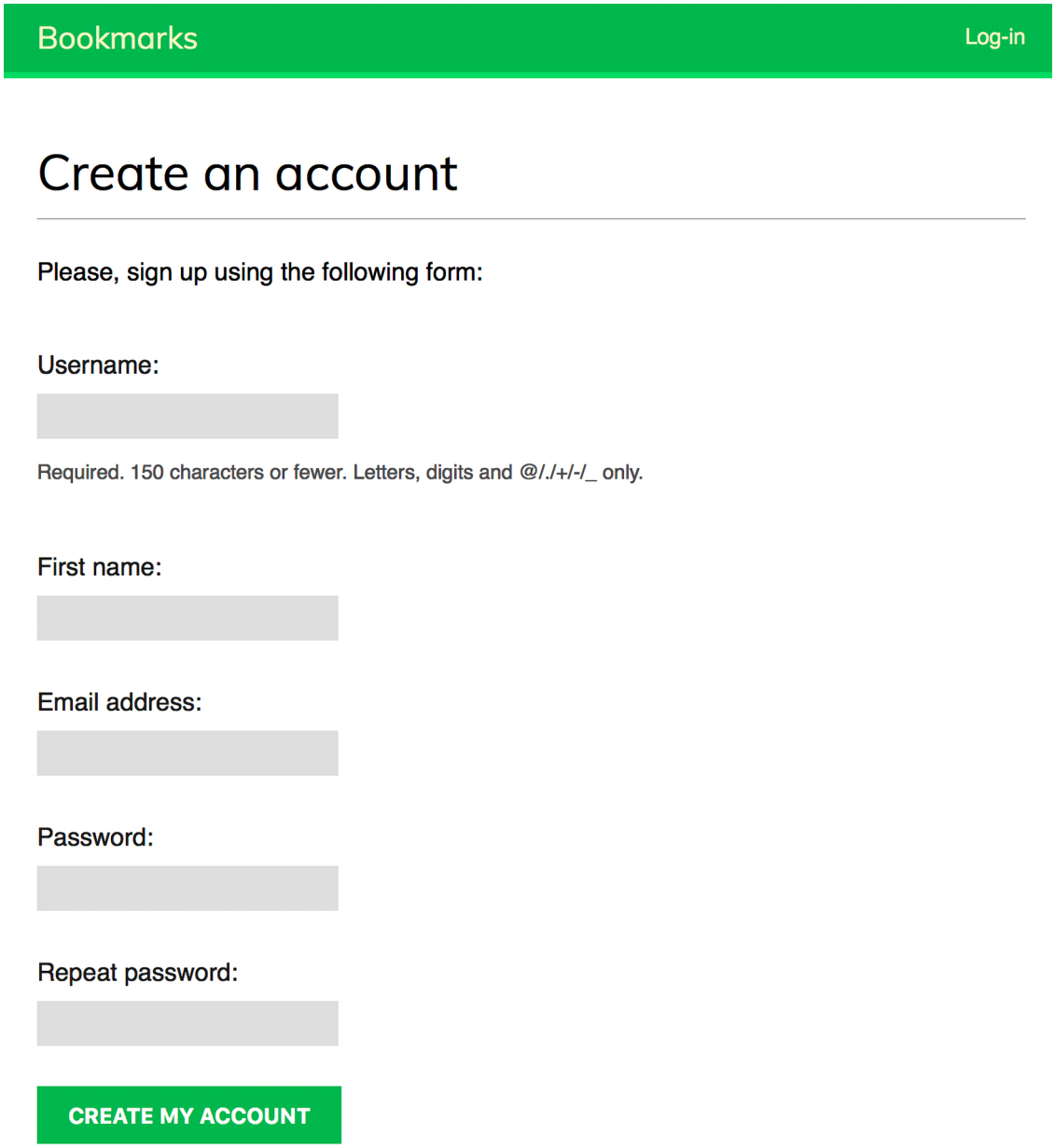


Figure 4.16: The account creation form

Fill in the details for a new user and click on the **CREATE MY ACCOUNT** button.

If all fields are valid, the user will be created, and you will see the following success message:

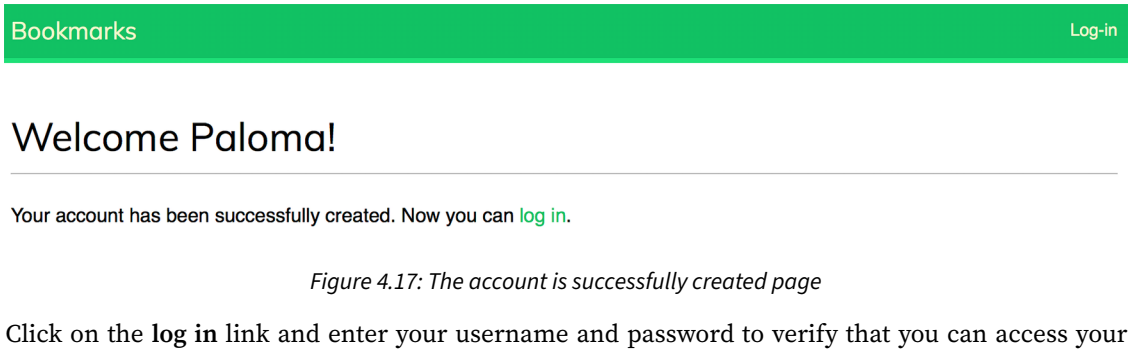


Figure 4.17: The account is successfully created page

Click on the **log in** link and enter your username and password to verify that you can access your newly created account.

Let's add a link to register on the login template. Edit the `registration/login.html` template and find the following line:

```
<p>Please, use the following form to log-in:</p>
```

Replace it with the following lines:

```
<p>
  Please, use the following form to log-in.
  If you don't have an account <a href="{% url "register" %}">register here</
a>.
</p>
```

Open `http://127.0.0.1:8000/account/login/` in your browser. The page should now look as follows:

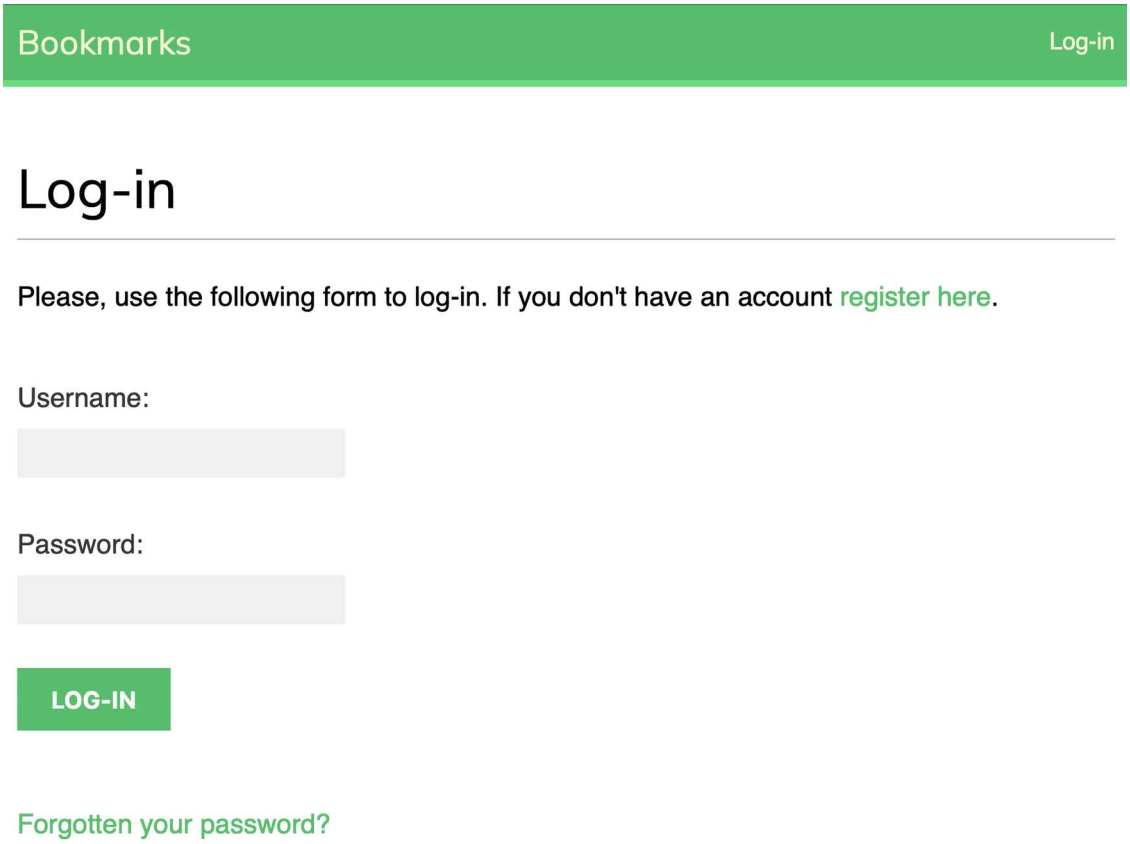


Figure 4.18: The Log-in page including a link to register

We have made the registration page accessible from the **Log-in** page.

Extending the user model

When dealing with user accounts, you will find that the `User` model of the Django authentication framework is suitable for most common cases. However, the standard `User` model comes with a limited set of fields. You may want to extend it with additional information that is relevant to your application.

A simple way to extend the `User` model is by creating a profile model that contains a one-to-one relationship with the Django `User` model, and any additional fields. A one-to-one relationship is similar to a `ForeignKey` field with the parameter `unique=True`. The reverse side of the relationship is an implicit one-to-one relationship with the related model instead of a manager for multiple elements. From each side of the relationship, you access a single related object.

Edit the `models.py` file of your `account` application and add the following code highlighted in bold:

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

class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                                on_delete=models.CASCADE)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='users/%Y/%m/%d/',
                              blank=True)

    def __str__(self):
        return f'Profile of {self.user.username}'
```

In order to keep your code generic, use the `get_user_model()` method to retrieve the user model and the `AUTH_USER_MODEL` setting to refer to it when defining a model's relationship with the user model, instead of referring to the `auth` user model directly. You can read more information about this at `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model`.

Our user profile will include the user's date of birth and an image of the user.

The one-to-one field `user` will be used to associate profiles with users. With `on_delete=models.CASCADE`, we force the deletion of the related `Profile` object when a `User` object gets deleted.

The `date_of_birth` field is a `DateField`. We have made this field optional with `blank=True`, and we allow `null` values with `null=True`.

The `photo` field is an `ImageField`. We have made this field optional with `blank=True`. An `ImageField` field manages the storage of image files. It validates the file provided is a valid image, stores the image file in the directory indicated with the `upload_to` parameter, and stores the relative path to the file in the related database field. An `ImageField` field is translated to a `VARHAR(100)` column in the database by default. A blank string will be stored if the value is left empty.

Installing Pillow and serving media files

We need to install the Pillow library to manage images. Pillow is the de facto standard library for image processing in Python. It supports multiple image formats and provides powerful image processing functions. Pillow is required by Django to handle images with `ImageField`.

Install Pillow by running the following command from the shell prompt:

```
pip install Pillow==9.2.0
```

Edit the `settings.py` file of the project and add the following lines:

```
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

This will enable Django to manage file uploads and serve media files. `MEDIA_URL` is the base URL used to serve the media files uploaded by users. `MEDIA_ROOT` is the local path where they reside. Paths and URLs for files are built dynamically by prepending the project path or the media URL to them for portability.

Now, edit the main `urls.py` file of the `bookmarks` project and modify the code, as follows. New lines are highlighted in bold:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

We have added the `static()` helper function to serve media files with the Django development server during development (that is when the `DEBUG` setting is set to `True`).

The `static()` helper function is suitable for development but not for production use. Django is very inefficient at serving static files. Never serve your static files with Django in a production environment. You will learn how to serve static files in a production environment in *Chapter 17*, *Going Live*.

Creating migrations for the profile model

Open the shell and run the following command to create the database migration for the new model:

```
python manage.py makemigrations
```

You will get the following output:

```
Migrations for 'account':
  account/migrations/0001_initial.py
    - Create model Profile
```

Next, sync the database with the following command in the shell prompt:

```
python manage.py migrate
```

You will see an output that includes the following line:

```
Applying account.0001_initial... OK
```

Edit the `admin.py` file of the `account` application and register the `Profile` model in the administration site by adding the code in bold:

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

@admin.register(Profile)
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'date_of_birth', 'photo']
    raw_id_fields = ['user']
```

Run the development server using the following command from the shell prompt:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/` in your browser. Now you should be able to see the `Profile` model on the administration site of your project, as follows:

Figure 4.19: The ACCOUNT block on the administration site index page

Click on the **Add** link of the **Profiles** row. You will see the following form to add a new profile:

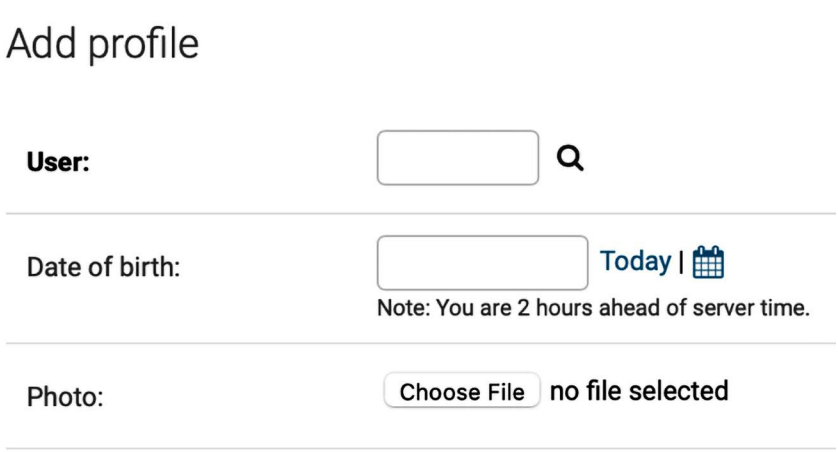


Figure 4.20: The Add profile form

Create a `Profile` object manually for each of the existing users in the database.

Next, we will let users edit their profiles on the website.

Edit the `forms.py` file of the `account` application and add the following lines highlighted in bold:

```
# ...
from .models import Profile

# ...

class UserEditForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ['first_name', 'last_name', 'email']

class ProfileEditForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['date_of_birth', 'photo']
```

These forms are as follows:

- `UserEditForm`: This will allow users to edit their first name, last name, and email, which are attributes of the built-in Django `User` model.
- `ProfileEditForm`: This will allow users to edit the profile data that is saved in the custom `Profile` model. Users will be able to edit their date of birth and upload an image for their profile picture.

Edit the `views.py` file of the `account` application and add the following lines highlighted in bold:

```
# ...
from .models import Profile

# ...

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Create a new user object but avoid saving it yet
            new_user = user_form.save(commit=False)
            # Set the chosen password
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Save the User object
            new_user.save()
            # Create the user profile
            Profile.objects.create(user=new_user)
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

When users register on the site, a `Profile` object will be created and associated with the `User` object created.

Now, we will let users edit their profiles.

Edit the `views.py` file of the `account` application and add the following code highlighted in bold:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
```

```
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm, \
                   UserEditForm, ProfileEditForm
from .models import Profile

# ...

@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile,
                                    data=request.POST,
                                    files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile)
    return render(request,
                  'account/edit.html',
                  {'user_form': user_form,
                   'profile_form': profile_form})
```

We have added the new `edit` view to allow users to edit their personal information. We have added the `login_required` decorator to the view because only authenticated users will be able to edit their profiles. For this view, we use two model forms: `UserEditForm` to store the data of the built-in `User` model and `ProfileEditForm` to store the additional personal data in the custom `Profile` model. To validate the data submitted, we call the `is_valid()` method of both forms. If both forms contain valid data, we save both forms by calling the `save()` method to update the corresponding objects in the database.

Add the following URL pattern to the `urls.py` file of the `account` application:

```
urlpatterns = [
    #...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
]
```

Finally, create a template for this view in the `templates/account/` directory and name it `edit.html`. Add the following code to it:

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

{% block title %}Edit your account{% endblock %}

{% block content %}
  <h1>Edit your account</h1>
  <p>You can edit your account using the following form:</p>
  <form method="post" enctype="multipart/form-data">
    {{ user_form.as_p }}
    {{ profile_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Save changes"></p>
  </form>
{% endblock %}
```

In the preceding code, we have added `enctype="multipart/form-data"` to the `<form>` HTML element to enable file uploads. We use an HTML form to submit both the `user_form` and `profile_form` forms.

Open the URL `http://127.0.0.1:8000/account/register/` and register a new user. Then, log in with the new user and open the URL `http://127.0.0.1:8000/account/edit/`. You should see the following page:

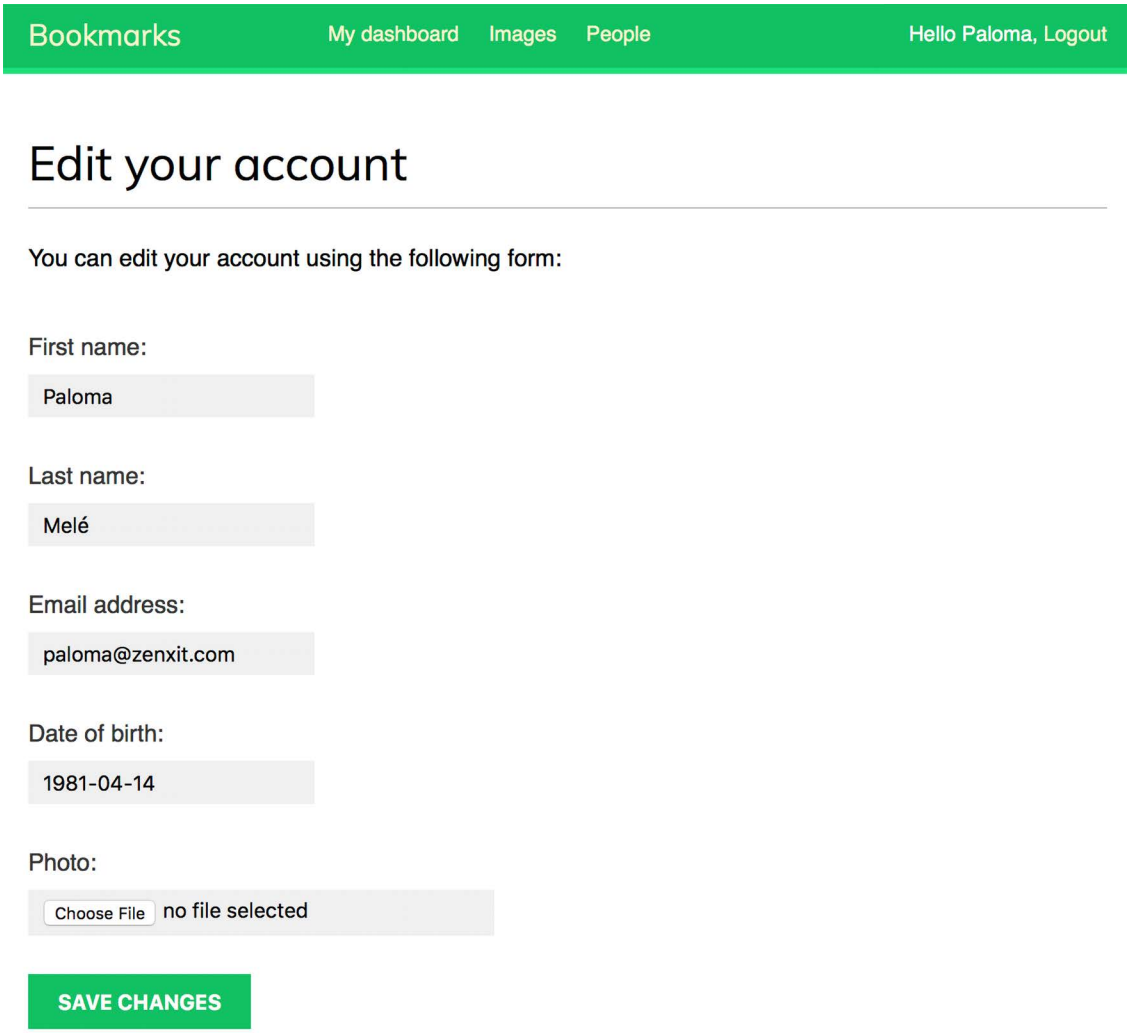


Figure 4.21: The profile edit form

You can now add the profile information and save the changes.

We will edit the dashboard template to include links to the edit profile and change password pages.

Open the `templates/account/dashboard.html` template and add the following lines highlighted in bold:

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

{% block title %}Dashboard{% endblock %}

{% block content %}
  <h1>Dashboard</h1>
  <p>
    Welcome to your dashboard. You can <a href="{% url "edit" %}">edit your
profile</a> or <a href="{% url "password_change" %}">change your password</a>.
  </p>
{% endblock %}
```

Users can now access the form to edit their profile from the dashboard. Open `http://127.0.0.1:8000/account/` in your browser and test the new link to edit a user's profile. The dashboard should now look like this:

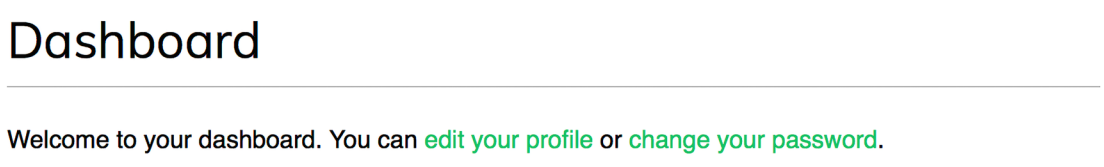


Figure 4.22: Dashboard page content, including links to edit a profile and change a password

Using a custom user model

Django also offers a way to substitute the `User` model with a custom model. The `User` class should inherit from Django's `AbstractUser` class, which provides the full implementation of the default user as an abstract model. You can read more about this method at `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#substituting-a-custom-user-model`.

Using a custom user model will give you more flexibility, but it might also result in more difficult integration with pluggable applications that interact directly with Django's `auth` user model.

Using the messages framework

When users are interacting with the platform, there are many cases where you might want to inform them about the result of specific actions. Django has a built-in messages framework that allows you to display one-time notifications to your users.

The messages framework is located at `django.contrib.messages` and is included in the default `INSTALLED_APPS` list of the `settings.py` file when you create new projects using `python manage.py startproject`. The settings file also contains the middleware `django.contrib.messages.middleware.MessageMiddleware` in the `MIDDLEWARE` setting.

The messages framework provides a simple way to add messages to users. Messages are stored in a cookie by default (falling back to session storage), and they are displayed and cleared in the next request from the user. You can use the messages framework in your views by importing the `messages` module and adding new messages with simple shortcuts, as follows:

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

You can create new messages using the `add_message()` method or any of the following shortcut methods:

- `success()`: Success messages to display when an action was successful
- `info()`: Informational messages
- `warning()`: A failure has not yet occurred but it may be imminent
- `error()`: An action was not successful or a failure occurred
- `debug()`: Debug messages that will be removed or ignored in a production environment

Let's add messages to the project. The messages framework applies globally to the project. We will use the base template to display any available messages to the client. This will allow us to notify the client with the results of any action on any page.

Open the `templates/base.html` template of the `account` application and add the following code highlighted in bold:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    ...
  </div>
  {% if messages %}
    <ul class="messages">
      {% for message in messages %}
        <li class="{{ message.tags }}">
          {{ message|safe }}
```

```
          <a href="#" class="close">x</a>
        </li>
      {% endfor %}
    </ul>
  {% endif %}
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
</body>
</html>
```

The messages framework includes the context processor `django.contrib.messages.context_processors.messages`, which adds a `messages` variable to the request context. You can find it in the `context_processors` list in the `TEMPLATES` setting of your project. You can use the `messages` variable in templates to display all existing messages to the user.

A context processor is a Python function that takes the `request` object as an argument and returns a dictionary that gets added to the request context. You will learn how to create your own context processors in *Chapter 8*, *Building an Online Shop*.

Let's modify the `edit` view to use the messages framework.

Edit the `views.py` file of the `account` application and add the following lines highlighted in bold:

```
# ...
from django.contrib import messages

# ...

@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile,
                                    data=request.POST,
                                    files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
```

```
            profile_form.save()
            messages.success(request, 'Profile updated '\
                                      'successfully')
        else:
            messages.error(request, 'Error updating your profile')
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile)
    return render(request,
                  'account/edit.html',
                  {'user_form': user_form,
                   'profile_form': profile_form})
```

A success message is generated when users successfully update their profile. If any of the forms contain invalid data, an error message is generated instead.

Open `http://127.0.0.1:8000/account/edit/` in your browser and edit the profile of the user. You should see the following message when the profile is successfully updated:

Figure 4.23: The successfully edited profile message

Enter an invalid date in the **Date of birth** field and submit the form again. You should see the following message:

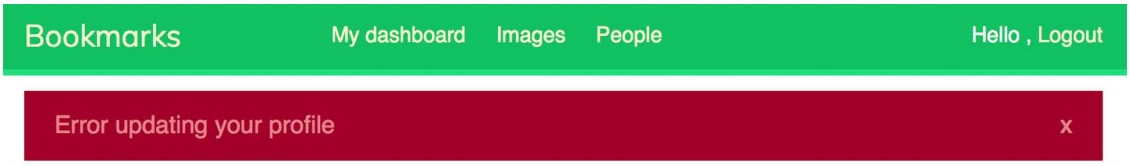


Figure 4.24: The error updating profile message

Generating messages to inform your users about the results of their actions is really straightforward. You can easily add messages to other views as well.

You can learn more about the messages framework at `https://docs.djangoproject.com/en/4.1/ref/contrib/messages/`.

Now that we've built all the functionality related to user authentication and profile editing, we will dig deeper into customizing authentication. We will learn how to build custom backend authentication so that users can log into the site using their email address.

Building a custom authentication backend

Django allows you to authenticate users against different sources. The `AUTHENTICATION_BACKENDS` setting includes a list of authentication backends available in the project. The default value of this setting is the following:

```
['django.contrib.auth.backends.ModelBackend']
```

The default `ModelBackend` authenticates users against the database using the `User` model of `django.contrib.auth`. This is suitable for most web projects. However, you can create custom backends to authenticate your users against other sources, such as a **Lightweight Directory Access Protocol** (**LDAP**) directory or any other system.

You can read more information about customizing authentication at `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#other-authentication-sources`.

Whenever the `authenticate()` function of `django.contrib.auth` is used, Django tries to authenticate the user against each of the backends defined in `AUTHENTICATION_BACKENDS` one by one, until one of them successfully authenticates the user. Only if all of the backends fail to authenticate will the user not be authenticated.

Django provides a simple way to define your own authentication backends. An authentication backend is a class that provides the following two methods:

- `authenticate()`: It takes the `request` object and user credentials as parameters. It has to return a `user` object that matches those credentials if the credentials are valid, or `None` otherwise. The `request` parameter is an `HttpRequest` object, or `None` if it's not provided to the `authenticate()` function.
- `get_user()`: It takes a user ID parameter and has to return a `user` object.

Creating a custom authentication backend is as simple as writing a Python class that implements both methods. Let's create an authentication backend to allow users to authenticate on the site using their email address instead of their username.

Create a new file inside the `account` application directory and name it `authentication.py`. Add the following code to it:

```
from django.contrib.auth.models import User

class EmailAuthBackend:
    """
    Authenticate using an e-mail address.
    """
    def authenticate(self, request, username=None, password=None):
        try:
            user = User.objects.get(email=username)
            if user.check_password(password):
```

```
                return user
            return None
        except (User.DoesNotExist, User.MultipleObjectsReturned):
            return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

The preceding code is a simple authentication backend. The `authenticate()` method receives a `request` object and the `username` and `password` optional parameters. We could use different parameters, but we use `username` and `password` to make our backend work with the authentication framework views right away. The preceding code works as follows:

- `authenticate()`: The user with the given email address is retrieved, and the password is checked using the built-in `check_password()` method of the user model. This method handles the password hashing to compare the given password with the password stored in the database. Two different QuerySet exceptions are captured: `DoesNotExist` and `MultipleObjectsReturned`. The `DoesNotExist` exception is raised if no user is found with the given email address. The `MultipleObjectsReturned` exception is raised if multiple users are found with the same email address. We will modify the registration and edit views later to prevent users from using an existing email address.
- `get_user()`: You get a user through the ID provided in the `user_id` parameter. Django uses the backend that authenticated the user to retrieve the `User` object for the duration of the user session. **pk** is a short for **primary key**, which is a unique identifier for each record in the database. Every Django model has a field that serves as its primary key. By default, the primary key is the automatically generated id field. The primary key can be also referred to as pk in the Django ORM. You can find more information about automatic primary key fields at `https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields`.

Edit the `settings.py` file of your project and add the following code:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

In the preceding setting, we keep the default `ModelBackend` that is used to authenticate with the username and password and include our own email-based authentication backend `EmailAuthBackend`.

Open `http://127.0.0.1:8000/account/login/` in your browser. Remember that Django will try to authenticate the user against each of the backends, so now you should be able to log in seamlessly using your username or email account.

The user credentials will be checked using `ModelBackend`, and if no user is returned, the credentials will be checked using `EmailAuthBackend`.

The order of the backends listed in the `AUTHENTICATION_BACKENDS` setting matters. If the same credentials are valid for multiple backends, Django will stop at the first backend that successfully authenticates the user.

Preventing users from using an existing email

The `User` model of the authentication framework does not prevent creating users with the same email address. If two or more user accounts share the same email address, we won't be able to discern which user is authenticating. Now that users can log in using their email address, we have to prevent users from registering with an existing email address.

We will now change the user registration form, to prevent multiple users from registering with the same email address.

Edit the `forms.py` file of the `account` application and add the following lines highlighted in bold to the `UserRegistrationForm` class:

```
class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'first_name', 'email']

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']

    def clean_email(self):
        data = self.cleaned_data['email']
        if User.objects.filter(email=data).exists():
            raise forms.ValidationError('Email already in use.')
        return data
```

We have added validation for the `email` field that prevents users from registering with an existing email address. We build a QuerySet to look up existing users with the same email address. We check whether there are any results with the `exists()` method. The `exists()` method returns `True` if the QuerySet contains any results, and `False` otherwise.

Now, add the following lines highlighted in bold to the `UserEditForm` class:

```
class UserEditForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ['first_name', 'last_name', 'email']

    def clean_email(self):
        data = self.cleaned_data['email']
        qs = User.objects.exclude(id=self.instance.id)\
                         .filter(email=data)
        if qs.exists():
            raise forms.ValidationError(' Email already in use.')
        return data
```

In this case, we have added validation for the `email` field that prevents users from changing their existing email address to an existing email address of another user. We exclude the current user from the QuerySet. Otherwise, the current email address of the user would be considered an existing email address, and the form won't validate.

Additional resources

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

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04`
- Built-in authentication views – `https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views`
- Authentication URL patterns – `https://github.com/django/django/blob/stable/3.0.x/django/contrib/auth/urls.py`
- How Django manages passwords and available password hashers – `https://docs.djangoproject.com/en/4.1/topics/auth/passwords/`
- Generic user model and the `get_user_model()` method – `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model`
- Using a custom user model – `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#substituting-a-custom-user-model`
- The Django messages framework – `https://docs.djangoproject.com/en/4.1/ref/contrib/messages/`

- Custom authentication sources – `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#other-authentication-sources`
- Automatic primary key fields – `https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields`

Summary

In this chapter, you learned how to build an authentication system for your site. You implemented all the necessary views for users to register, log in, log out, edit their password, and reset their password. You built a model for custom user profiles, and you created a custom authentication backend to let users log into your site using their email address.

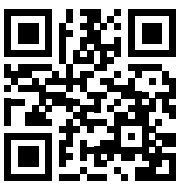
In the next chapter, you will learn how to implement social authentication on your site using Python Social Auth. Users will be able to authenticate with their Google, Facebook, or Twitter accounts. You will also learn how to serve the development server over HTTPS using Django Extensions. You will customize the authentication pipeline to create user profiles automatically.

Join us on Discord

Read this book alongside other users and the author.

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

`https://packt.link/django`



5

Implementing Social Authentication

In the previous chapter, you built user registration and authentication into your website. You implemented password change, reset, and recovery functionalities, and you learned how to create a custom profile model for your users.

In this chapter, you will add social authentication to your site using Facebook, Google, and Twitter. You will use Django Social Auth to implement social authentication using OAuth 2.0, the industry-standard protocol for authorization. You will also modify the social authentication pipeline to create a user profile for new users automatically.

This chapter will cover the following points:

- Adding social authentication with Python Social Auth
- Installing Django Extensions
- Running the development server through HTTPS
- Adding authentication using Facebook
- Adding authentication using Twitter
- Adding authentication using Google
- Creating a profile for users that register with social authentication

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter05`.

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

Adding social authentication to your site

Social authentication is a widely used feature that allows users to authenticate using their existing account of a service provider using **Single Sign-on** (**SSO**). The authentication process allows users to authenticate into the site using their existing account from social services like Google. In this section, we will add social authentication to the site using Facebook, Twitter, and Google.

To implement social authentication, we will use the **OAuth 2.0** industry-standard protocol for authorization. **OAuth** stands for *Open Authorization*. OAuth 2.0 is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user. Facebook, Twitter, and Google use the OAuth 2.0 protocol for authentication and authorization.

Python Social Auth is a Python module that simplifies the process of adding social authentication to your website. Using this module, you can let your users log in to your website using their accounts from other services. You can find the code for this module at `https://github.com/python-social-auth/social-app-django`.

This module comes with authentication backends for different Python frameworks, including Django. To install the Django package from the Git repository of the project, open the console and run the following command:

```
git+https://github.com/python-social-auth/social-app-django.
git@20fabcd7bd9a8a41910bc5c8ed1bd6ef2263b328
```

This will install Python Social Auth from a GitHub commit that works with Django 4.1. At the writing of this book the latest Python Social Auth release is not compatible with Django 4.1 but a newer compatible release might have been published.

Then add `social_django` to the `INSTALLED_APPS` setting in the `settings.py` file of the project as follows:

```
INSTALLED_APPS = [
    # ...
    'social_django',
]
```

This is the default application to add Python Social Auth to Django projects. Now run the following command to sync Python Social Auth models with your database:

```
python manage.py migrate
```

You should see that the migrations for the default application are applied as follows:

```
Applying social_django.0001_initial... OK
Applying social_django.0002_add_related_name... OK
...
Applying social_django.0011_alter_id_fields... OK
```

Python Social Auth includes authentication backends for multiple services. You can find the list with all available backends at `https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends`.

We will add social authentication to our project, allowing our users to authenticate with the Facebook, Twitter, and Google backends.

First, we need to add the social login URL patterns to the project.

Open the main `urls.py` file of the `bookmarks` project and include the `social_django` URL patterns as follows. New lines are highlighted in bold:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
         include('social_django.urls', namespace='social')),
]
```

Our web application is currently accessible via the localhost IP to `127.0.0.1` or using the `localhost` hostname. Several social services will not allow redirecting users to `127.0.0.1` or `localhost` after successful authentication; they expect a domain name for the URL redirect. First, we need to use a domain name to make social authentication work. Fortunately, we can simulate serving our site under a domain name in our local machine.

Locate the `hosts` file of your machine. If you are using Linux or macOS, the `hosts` file is located at `/etc/hosts`. If you are using Windows, the `hosts` file is located at `C:\Windows\System32\Drivers\etc\hosts`.

Edit the `hosts` file of your machine and add the following line to it:

```
127.0.0.1 mysite.com
```

This will tell your computer to point the `mysite.com` hostname to your own machine.

Let's verify that the hostname association worked. Run the development server using the following command from the shell prompt:

```
python manage.py runserver
```

Open `http://mysite.com:8000/account/login/` in your browser. You will see the following error:

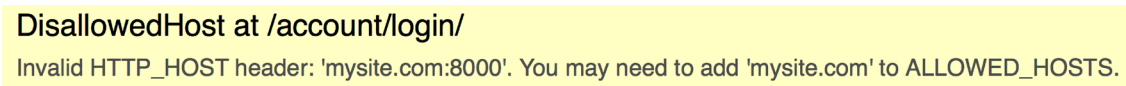


Figure 5.1: The invalid host header message

Django controls the hosts that can serve the application using the `ALLOWED_HOSTS` setting. This is a security measure to prevent HTTP host header attacks. Django will only allow the hosts included in this list to serve the application.

You can learn more about the `ALLOWED_HOSTS` setting at `https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts`.

Edit the `settings.py` file of the project and modify the `ALLOWED_HOSTS` setting as follows. New code is highlighted in bold:

```
ALLOWED_HOSTS = ['mysite.com', 'localhost', '127.0.0.1']
```

Besides the `mysite.com` host, we have explicitly included `localhost` and `127.0.0.1`. This allows access to the site through `localhost` and `127.0.0.1`, which is the default Django behavior when `DEBUG` is `True` and `ALLOWED_HOSTS` is empty.

Open `http://mysite.com:8000/account/login/` again in your browser. Now, you should see the login page of the site instead of an error.

Running the development server through HTTPS

Some of the social authentication methods we are going to use require an HTTPS connection. The **Transport Layer Security** (**TLS**) protocol is the standard for serving websites through a secure connection. The TLS predecessor is the **Secure Sockets Layer** (**SSL**).

Although SSL is now deprecated, in multiple libraries and online documentation you will find references to both the terms TLS and SSL. The Django development server is not able to serve your site through HTTPS, since that is not its intended use. To test the social authentication functionality serving the site through HTTPS, we are going to use the RunServerPlus extension of the package Django Extensions. Django Extensions is a third-party collection of custom extensions for Django. Please note that you should never use this to serve your site in a real environment; this is only a development server.

Use the following command to install Django Extensions:

```
pip install git+https://github.com/django-extensions/django-extensions.
git@25a41d8a3ecb24c009c5f4cac6010a091a3c91c8
```

This will install Django Extensions from a GitHub commit that includes support for Django 4.1. At the writing of this book the latest Django Extensions release is not compatible with Django 4.1 but a newer compatible release might have been published.

You will need to install Werkzeug, which contains a debugger layer required by the RunServerPlus extension of Django Extensions. Use the following command to install Werkzeug:

```
pip install werkzeug==2.2.2
```

Finally, use the following command to install pyOpenSSL, which is required to use the SSL/TLS functionality of RunServerPlus:

```
pip install pyOpenSSL==22.0.0
```

Edit the `settings.py` file of your project and add Django Extensions to the `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    # ...
    'django_extensions',
]
```

Now, use the management command `runserver_plus` provided by Django Extensions to run the development server, as follows:

```
python manage.py runserver_plus --cert-file cert.crt
```

We have provided a file name to the `runserver_plus` command for the SSL/TLS certificate. Django Extensions will generate a key and certificate automatically.

Open `https://mysite.com:8000/account/login/` in your browser. Now you are accessing your site through HTTPS. Note we are now using `https://` instead of `http://`.

Your browser will show a security warning because you are using a self-generated certificate instead of a certificate trusted by a **Certification Authority** (CA).

If you are using Google Chrome, you will see the following screen:

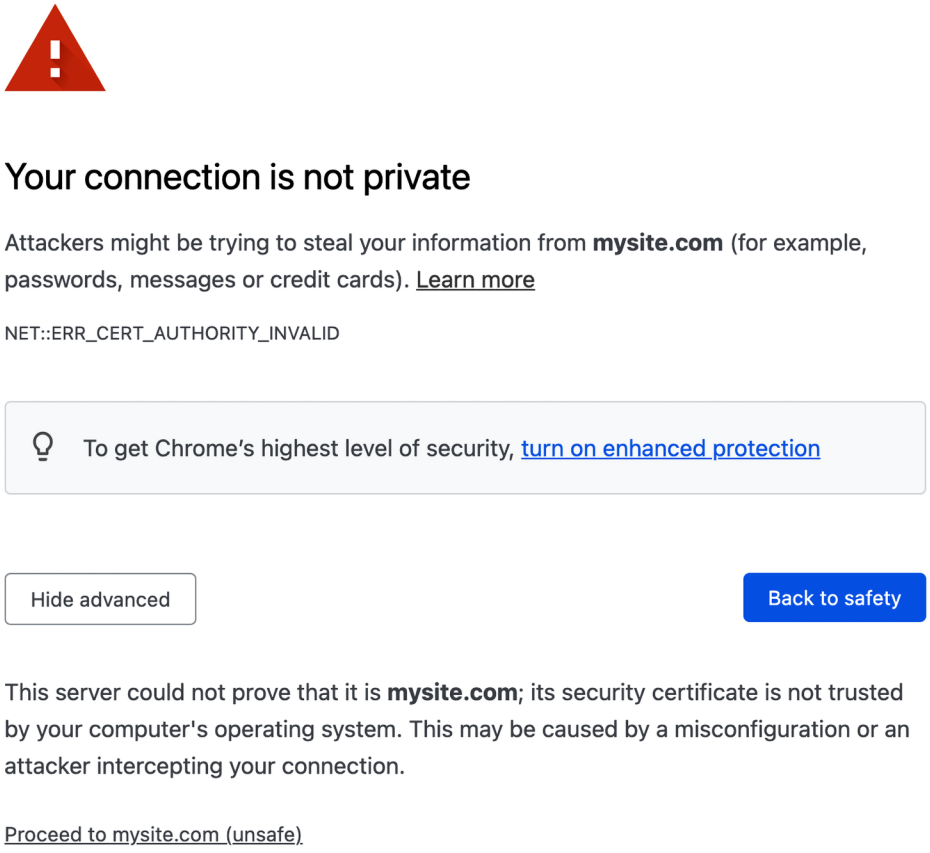


Figure 5.2: The safety error in Google Chrome

In this case, click on **Advanced** and then click on **Proceed to 127.0.0.1 (unsafe)**.

If you are using Safari, you will see the following screen:

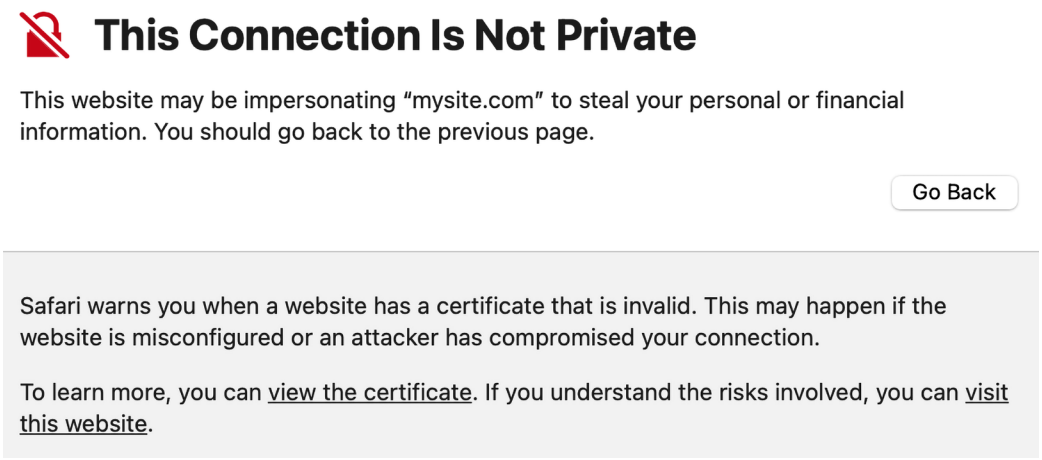


Figure 5.3: The safety error in Safari

In this case, click on **Show details** and then click on **visit this website**.

If you are using Microsoft Edge, you will see the following screen:

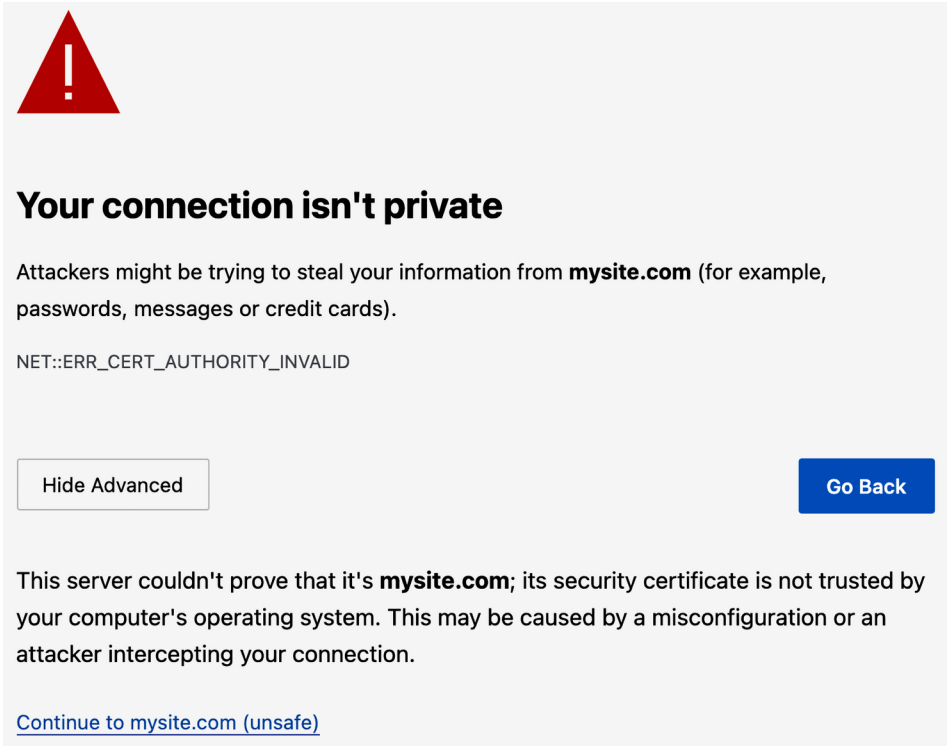


Figure 5.4: The safety error in Microsoft Edge

In this case, click on **Advanced** and then on **Continue to mysite.com (unsafe)**.

If you are using any other browser, access the advanced information displayed by your browser and accept the self-signed certificate so that your browser trusts the certificate.

You will see that the URL starts with `https://` and in some cases a lock icon that indicates that the connection is secure. Some browsers might display a broken lock icon because you are using a self-signed certificate instead of a trusted one. That won't be a problem for our tests:

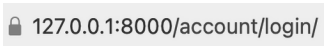


Figure 5.5: The URL with the secured connection icon

Django Extensions includes many other interesting tools and features. You can find more information about this package at `https://django-extensions.readthedocs.io/en/latest/`.

You can now serve your site through HTTPS during development to test social authentication with Facebook, Twitter, and Google.

Authentication using Facebook

To use Facebook authentication to log in to your site, add the following line highlighted in bold to the `AUTHENTICATION_BACKENDS` setting in the `settings.py` file of your project:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
    'social_core.backends.facebook.FacebookOAuth2',
]
```

You will need a Facebook developer account and you will need to create a new Facebook application.

Open `https://developers.facebook.com/apps/` in your browser. After creating a Facebook developer account, you will see a site with the following header:

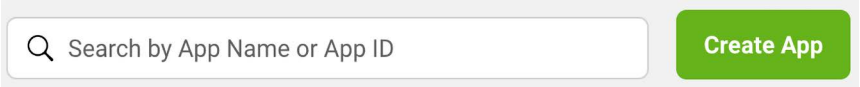


Figure 5.6: The Facebook developer portal header

Click on **Create App**.

You will see the following form to choose an application type:

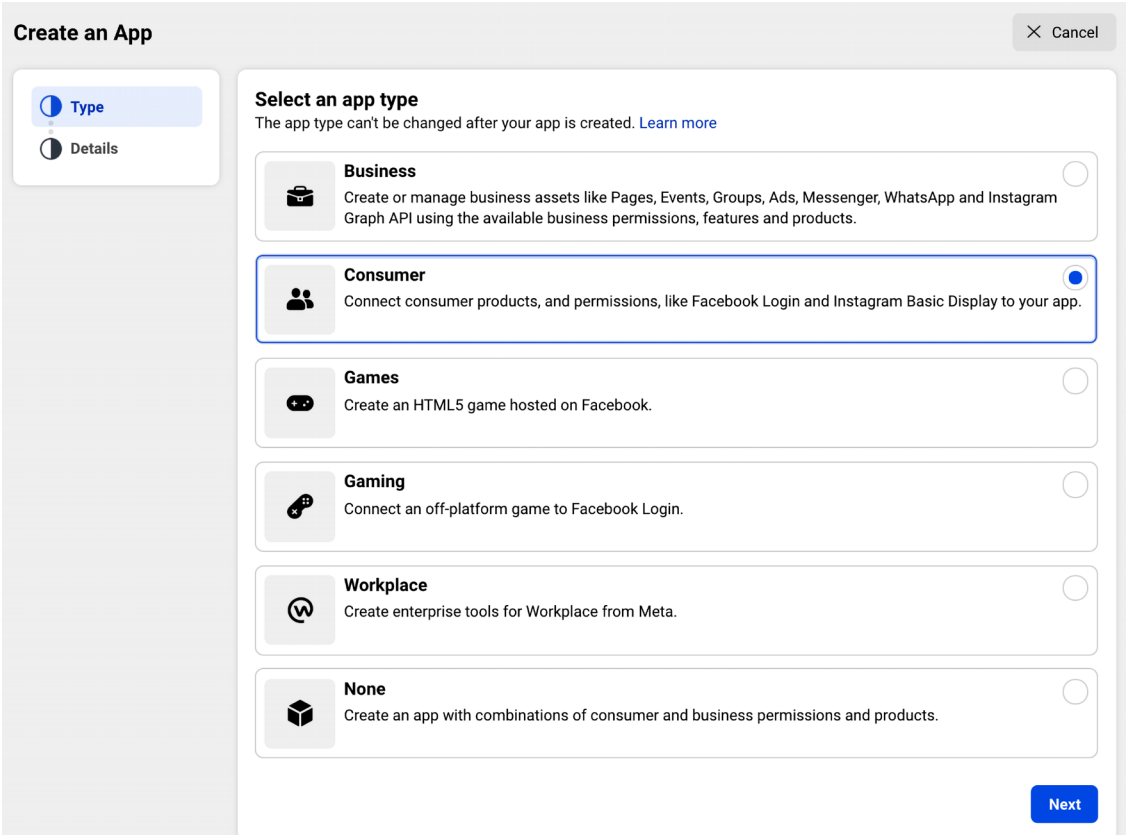


Figure 5.7: The Facebook create app form to select an application type

Under **Select an app type,** choose **Consumer** and click on **Next.**

You will see the following form to create a new application:

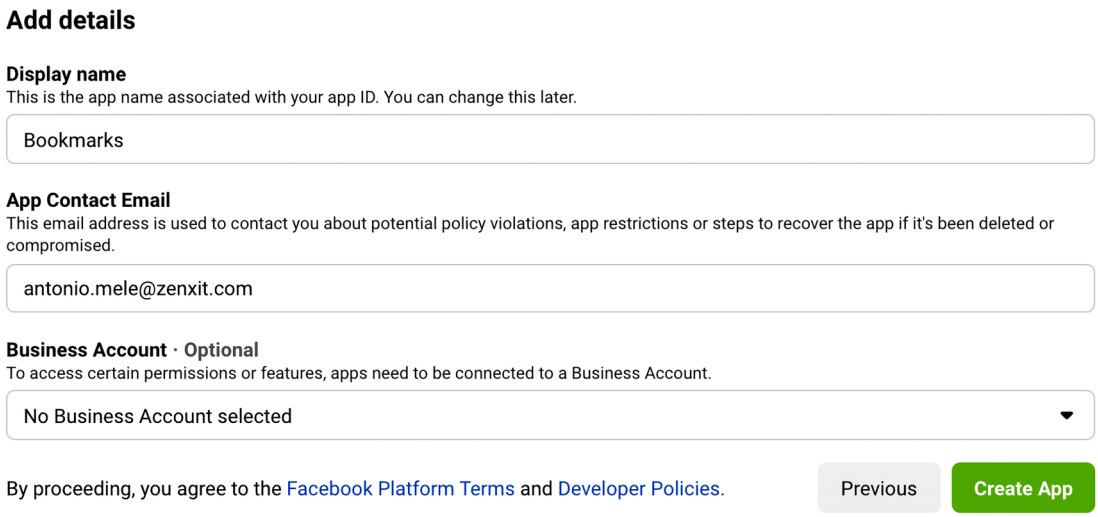


Figure 5.8: The Facebook form for application details

Enter `Bookmarks` as the **Display name**, add a contact email address, and click on **Create App**.

You will see the dashboard for your new application that displays different services that you can configure for the app. Look for the following **Facebook Login** box and click on **Set Up**:

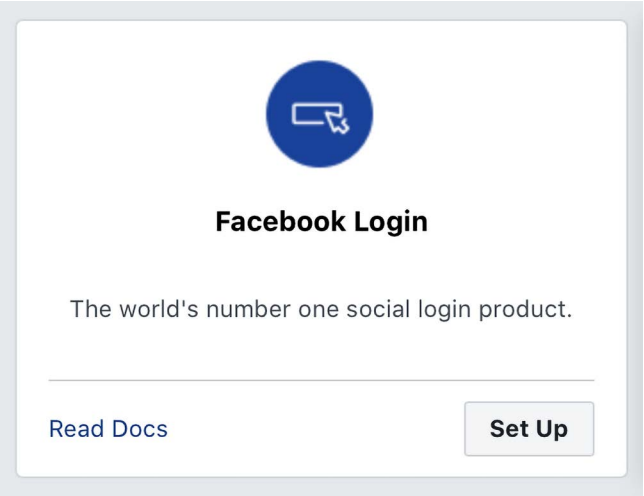


Figure 5.9: The Facebook login product block

You will be asked to choose the platform, as follows:

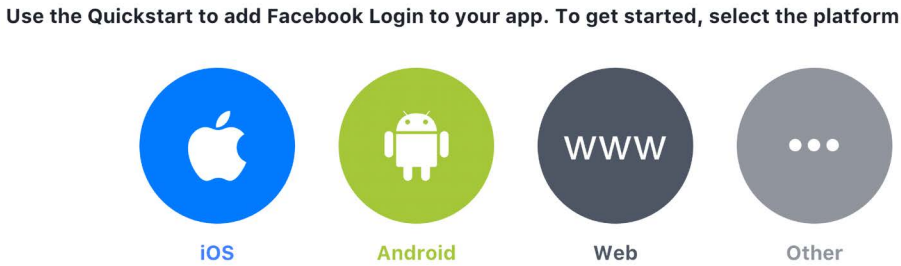


Figure 5.10: Platform selection for Facebook login

Select the **Web** platform. You will see the following form:

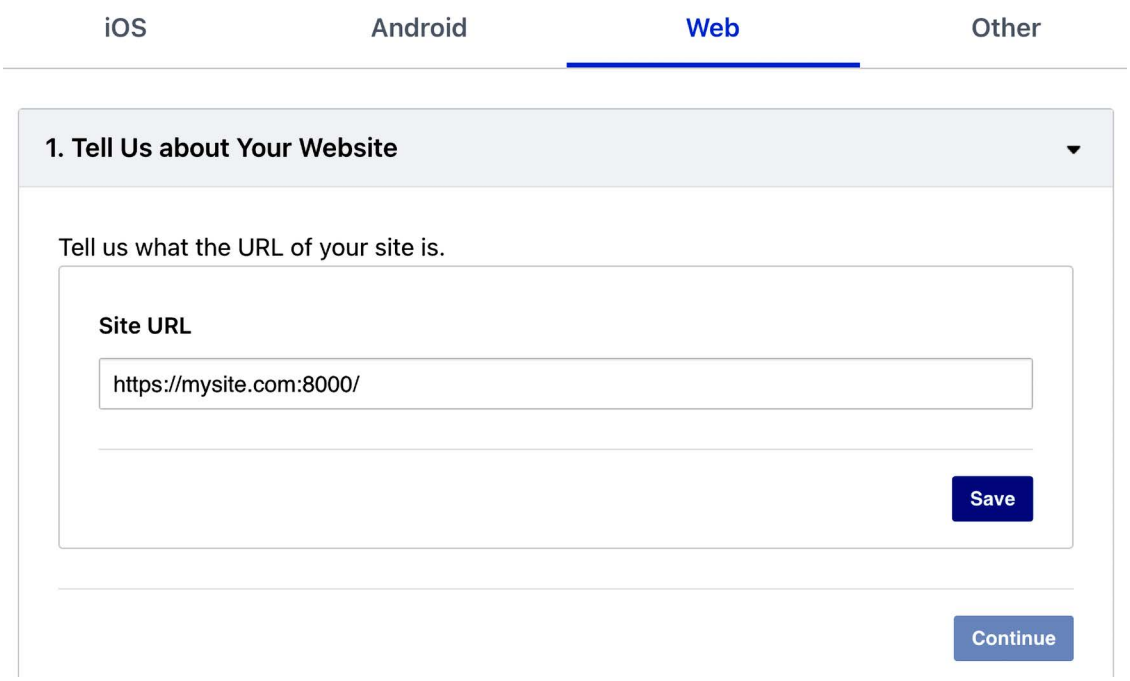


Figure 5.11: Web platform configuration for Facebook login

Enter `https://mysite.com:8000/` under **Site URL** and click the **Save** button. Then click **Continue**. You can skip the rest of the quick start process.

In the left-hand menu, click on **Settings** and then on **Basic**, as follows:

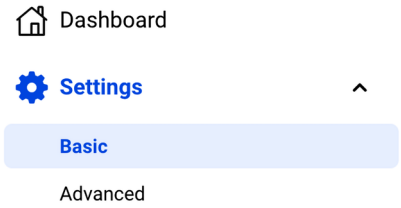


Figure 5.12: Facebook developer portal sidebar menu

You will see a form with data similar to the following one:

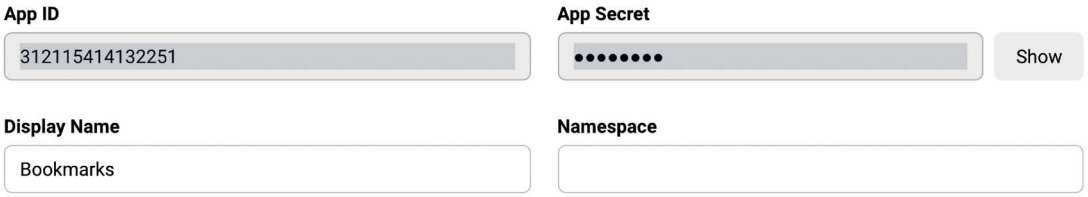


Figure 5.13: Application details for the Facebook application

Copy the **App ID** and **App Secret** keys and add them to the `settings.py` file of your project, as follows:

```
SOCIAL_AUTH_FACEBOOK_KEY = 'XXX' # Facebook App ID
SOCIAL_AUTH_FACEBOOK_SECRET = 'XXX' # Facebook App Secret
```

Optionally, you can define a `SOCIAL_AUTH_FACEBOOK_SCOPE` setting with the extra permissions you want to ask Facebook users for:

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

Now, go back to the Facebook developer portal and click on **Settings**. Add `mysite.com` under **App Domains**, as follows:

Figure 5.14: Allowed domains for the Facebook application

You have to enter a public URL for the **Privacy Policy URL** and another one for the **User Data Deletion Instructions URL**. The following is an example using the Wikipedia page URL for *Privacy Policy*. Please note that you should use a valid URL:

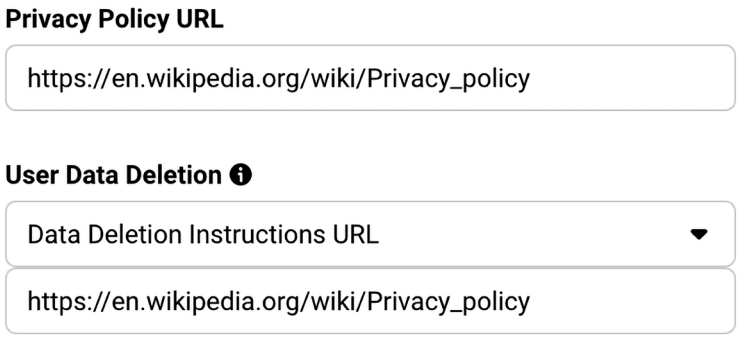


Figure 5.15: Privacy policy and user data deletion instructions URLs for the Facebook application

Click on **Save Changes**. Then, in the left-hand menu under **Products**, click on **Facebook Login** and then **Settings**, as shown here:

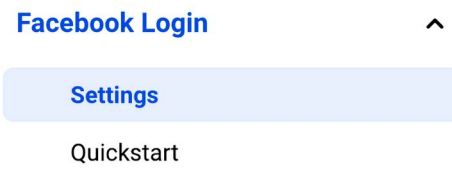


Figure 5.16: The Facebook login menu

Ensure that only the following settings are active:

- **Client OAuth Login**
- **Web OAuth Login**
- **Enforce HTTPS**
- **Embedded Browser OAuth Login**
- **Used Strict Mode for Redirect URIs**

Enter `https://mysite.com:8000/social-auth/complete/facebook/` under **Valid OAuth Redirect URIs**. The selection should look like this:

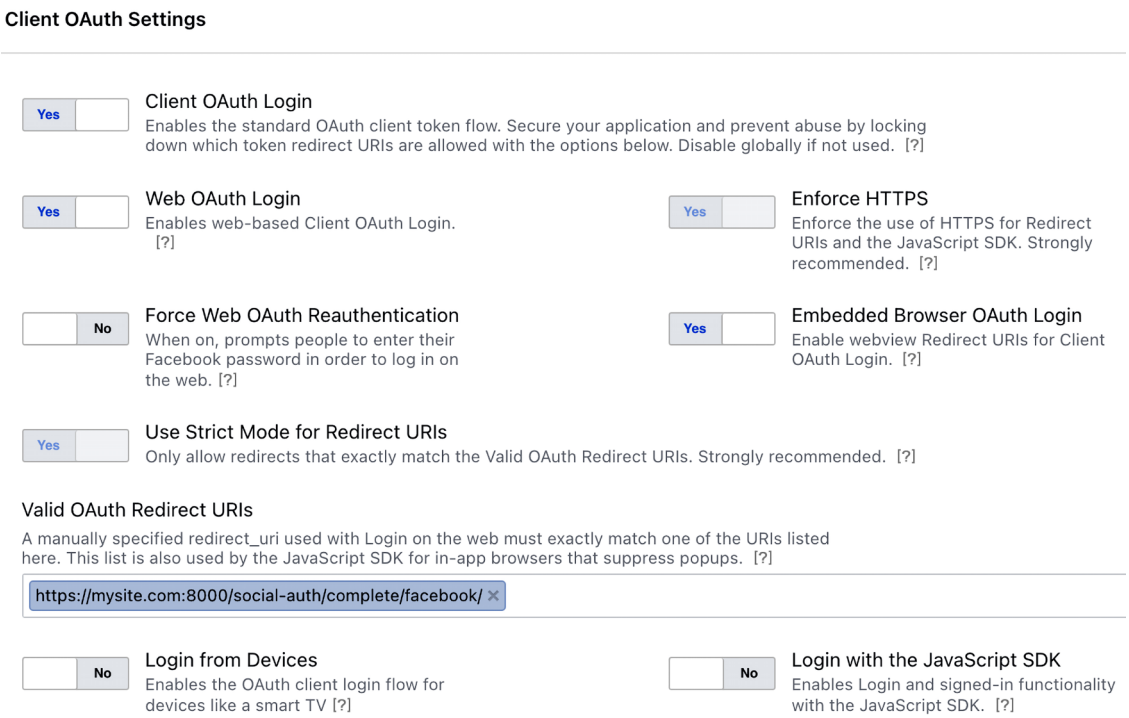


Figure 5.17: Client OAuth settings for Facebook login

Open the `registration/login.html` template of the `account` application and append the following code highlighted in bold at the bottom of the `content` block:

```
{% block content %}
  ...
  <div class="social">
    <ul>
      <li class="facebook">
        <a href="{% url "social:begin" "facebook" %}">
          Sign in with Facebook
        </a>
      </li>
    </ul>
  </div>
{% endblock %}
```

Use the management command `runserver_plus` provided by Django Extensions to run the development server, as follows:

```
python manage.py runserver_plus --cert-file cert.crt
```

Open `https://mysite.com:8000/account/login/` in your browser. The login page will look now as follows:

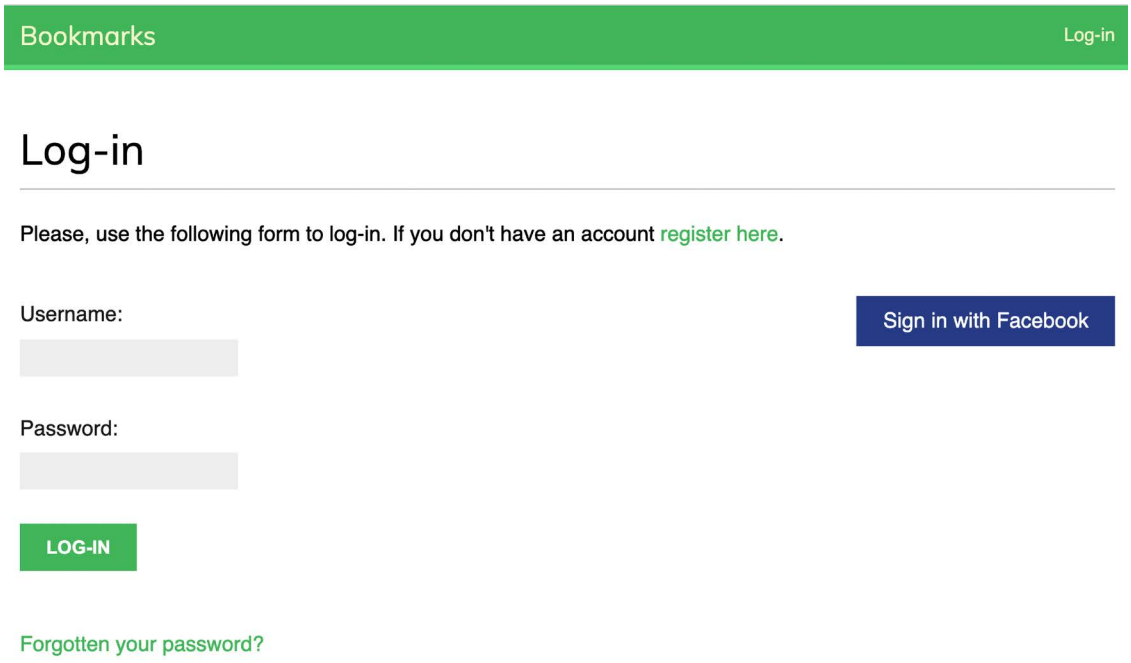


Figure 5.18: The login page including the button for Facebook authentication

Click on the **Sign in with Facebook** button. You will be redirected to Facebook, and you will see a modal dialog asking for your permission to let the *Bookmarks* application access your public Facebook profile:

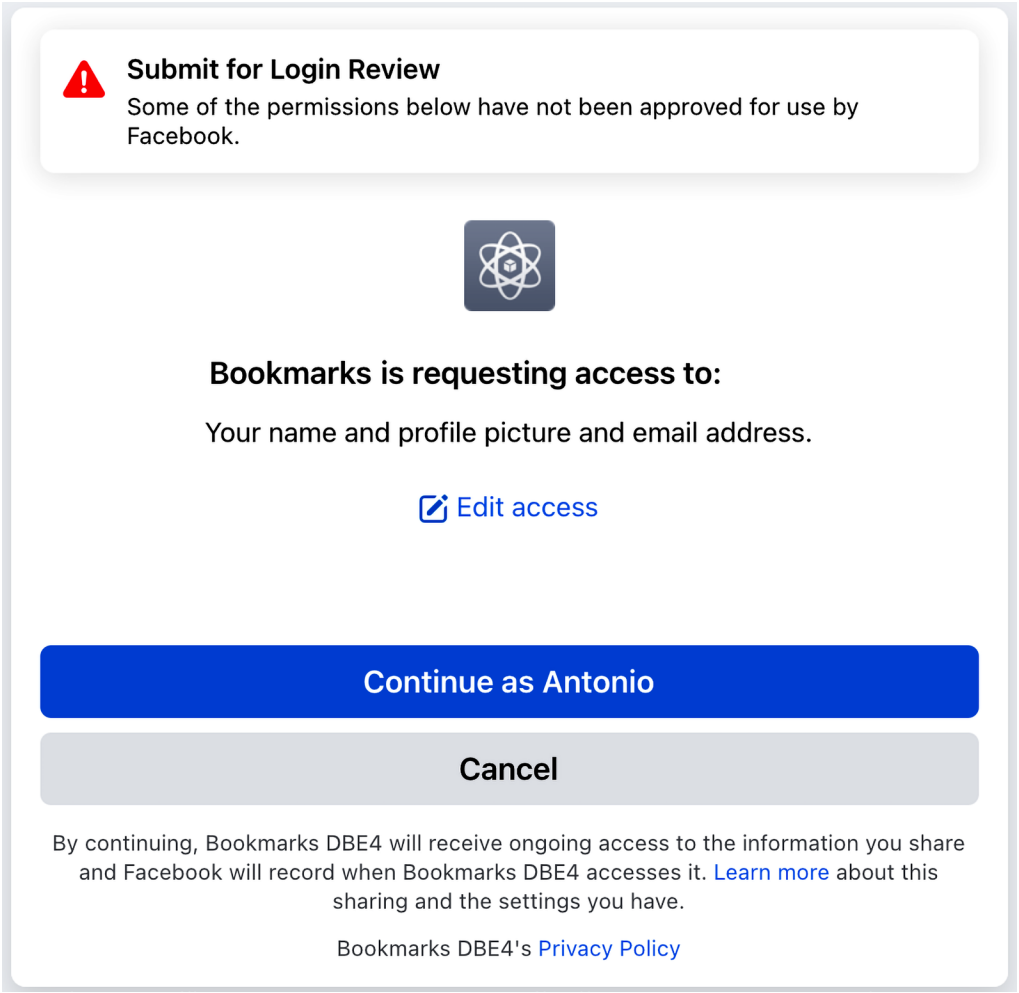


Figure 5.19: The Facebook modal dialog to grant application permissions

You will see a warning indicating that you need to submit the application for login review. Click on the **Continue as ...** button.

You will be logged in and redirected to the dashboard page of your site. Remember that you have set this URL in the `LOGIN_REDIRECT_URL` setting. As you can see, adding social authentication to your site is pretty straightforward.

Authentication using Twitter

For social authentication using Twitter, add the following line highlighted in bold to the `AUTHENTICATION_BACKENDS` setting in the `settings.py` file of your project:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
    'social_core.backends.facebook.FacebookOAuth2',
    'social_core.backends.twitter.TwitterOAuth',
]
```

You need a Twitter developer account. Open `https://developer.twitter.com/` in your browser and click on **Sign up**.

After creating a Twitter developer account, access the Developer Portal Dashboard at `https://developer.twitter.com/en/portal/dashboard`. The dashboard should look as follows:

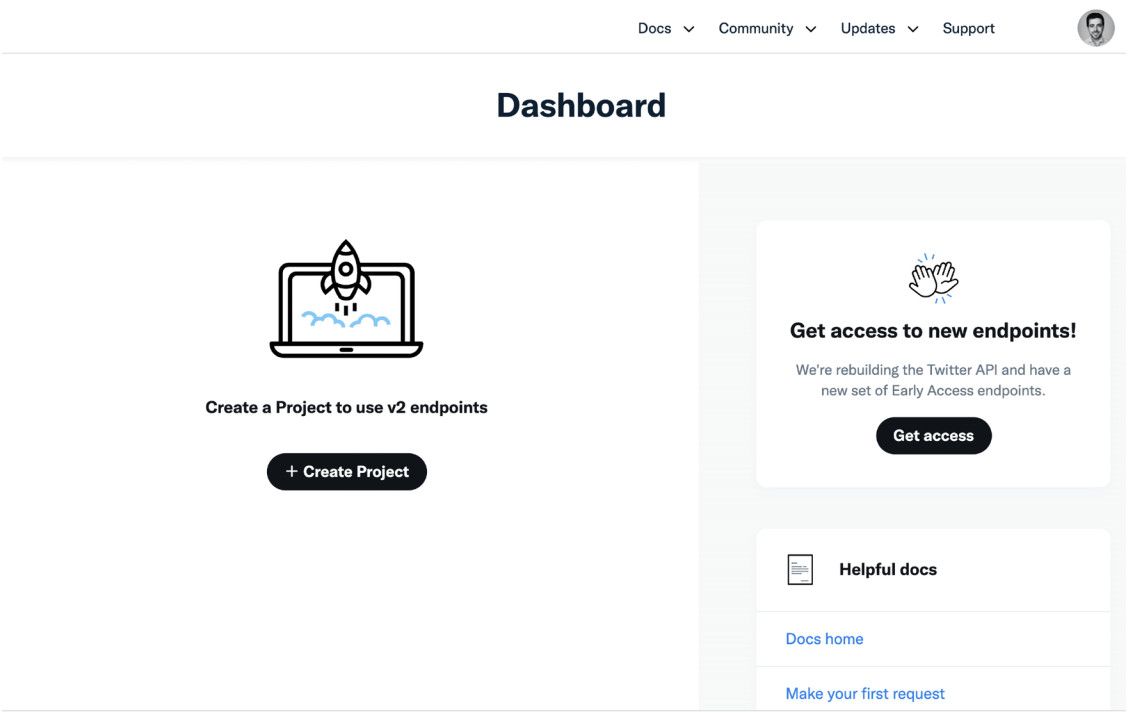


Figure 5.20: Twitter developer portal dashboard

Click on the **Create Project** button. You will see the following screen:

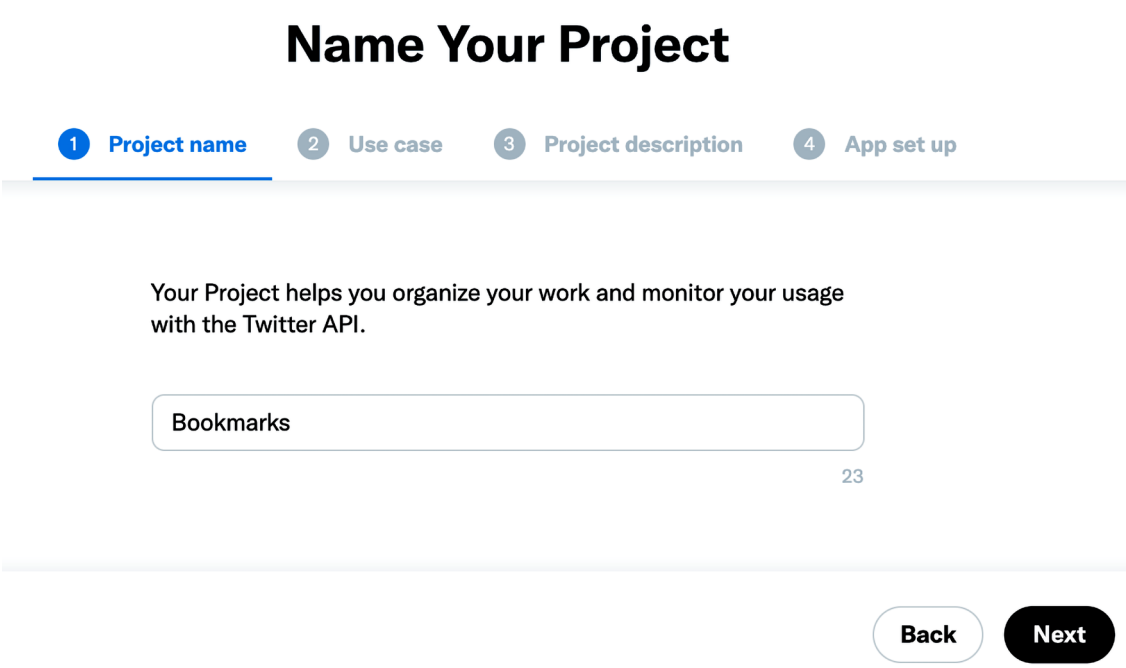


Figure 5.21: Twitter create project screen – Project name

Enter `Bookmarks` for the **Project name** and click on **Next**. You will see the following screen:

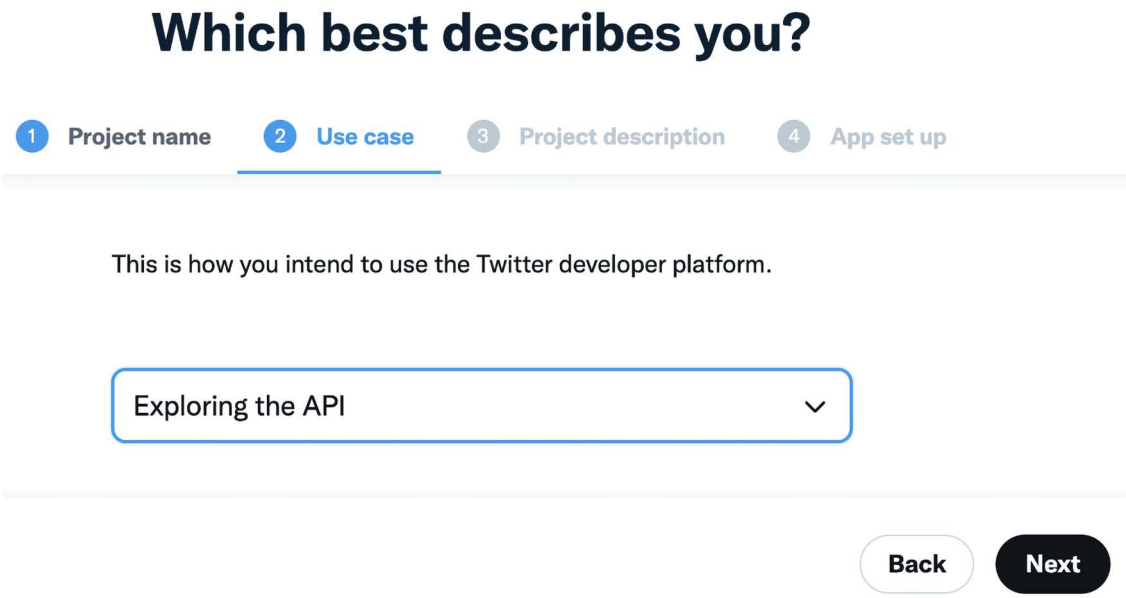


Figure 5.22: Twitter create project screen – Use case

Under **Use case**, select **Exploring the API** and click on **Next**. You can choose any other use case; it won't affect the configuration. Then you will see the following screen:

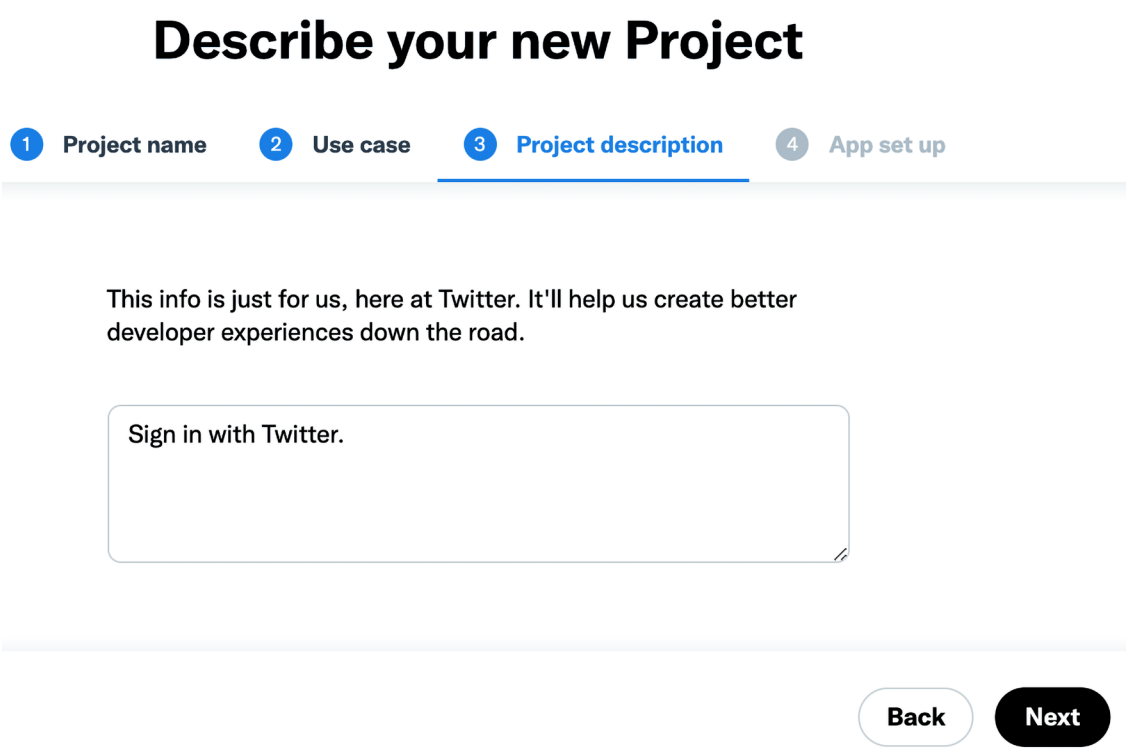


Figure 5.23: Twitter create project screen – Project description

Enter a short description for your project and click on **Next.** The project is now created, and you will see the following screen:

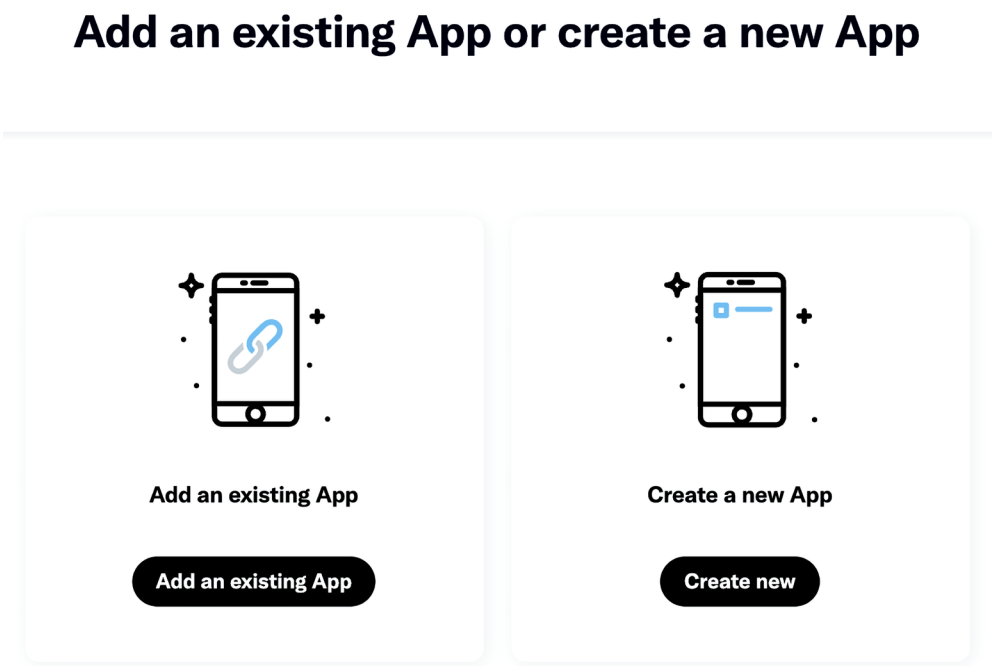


Figure 5.24: Twitter application configuration

We will create a new application. Click on **Create new**. You will see the following screen to configure the new application:

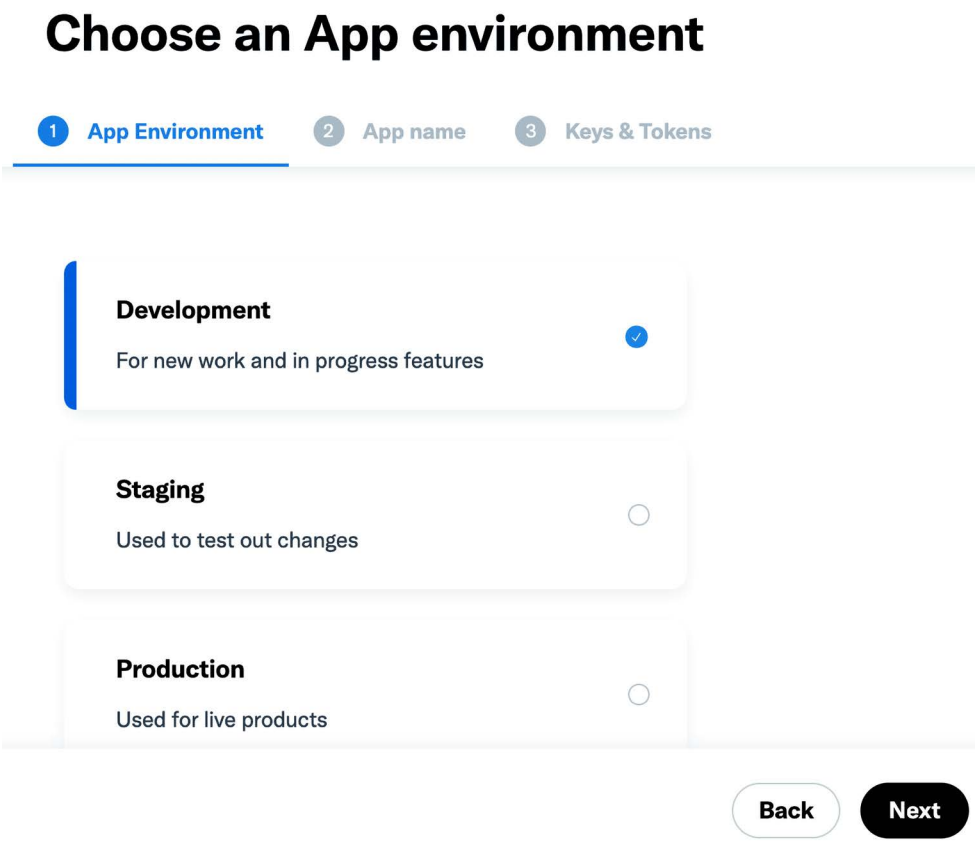


Figure 5.25: Twitter application configuration - environment selection

Under **App Environment**, select **Development** and click on **Next**. We are creating a development environment for the application. You will see the following screen:

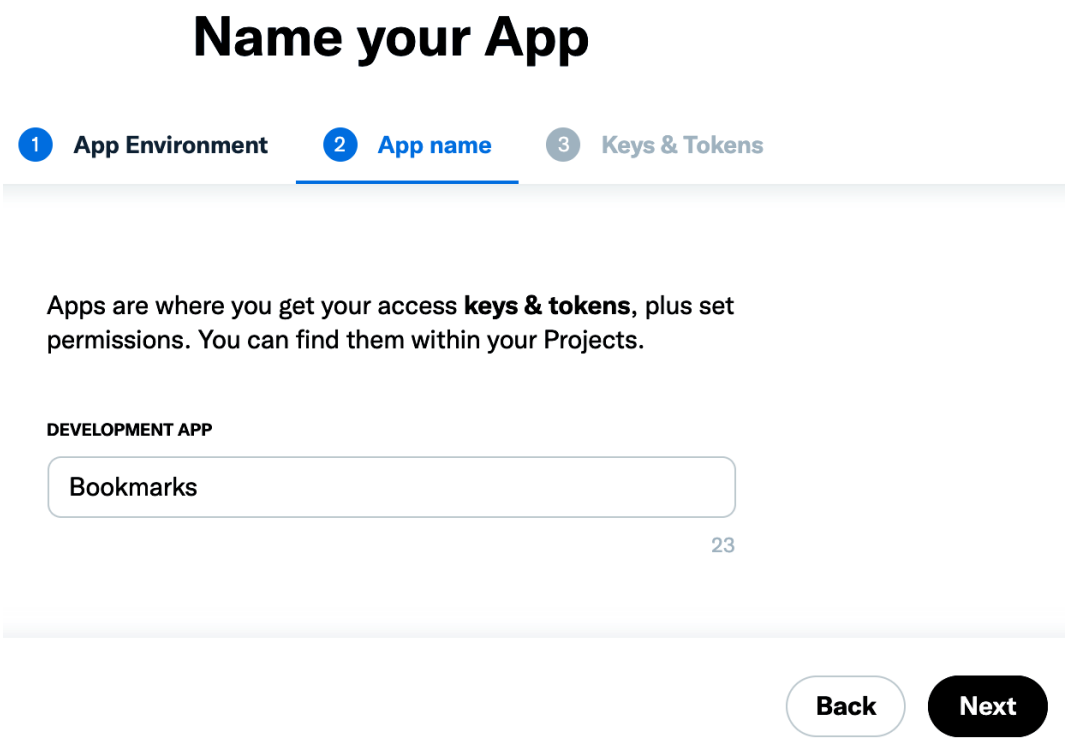


Figure 5.26: Twitter application configuration – App name

Under **App name**, enter `Bookmarks` followed by a suffix. Twitter won't allow you to use the name of an existing developer app within Twitter, so you need to enter a name that might be available. Click **Next**. Twitter will show you an error if the name you try to use for your app is already taken.

After choosing a name that is available, you will see the following screen:

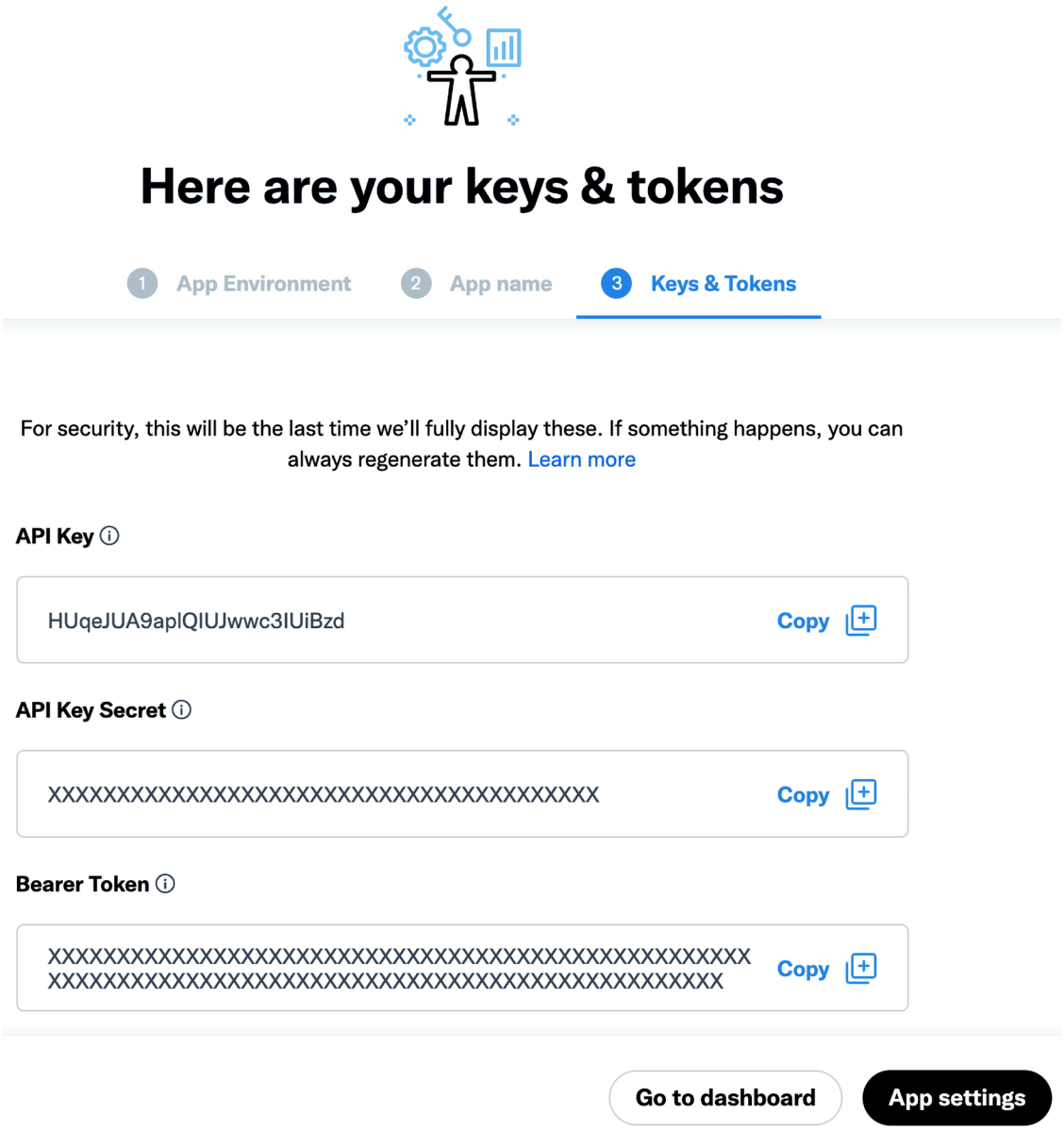


Figure 5.27: Twitter application configuration – generated API keys

Copy the **API Key** and **API Key Secret** into the following settings in the `settings.py` file of your project:

```
SOCIAL_AUTH_TWITTER_KEY = 'XXX' # Twitter API Key
SOCIAL_AUTH_TWITTER_SECRET = 'XXX' # Twitter API Secret
```

Then click on **App settings**. You will see a screen that includes the following section:

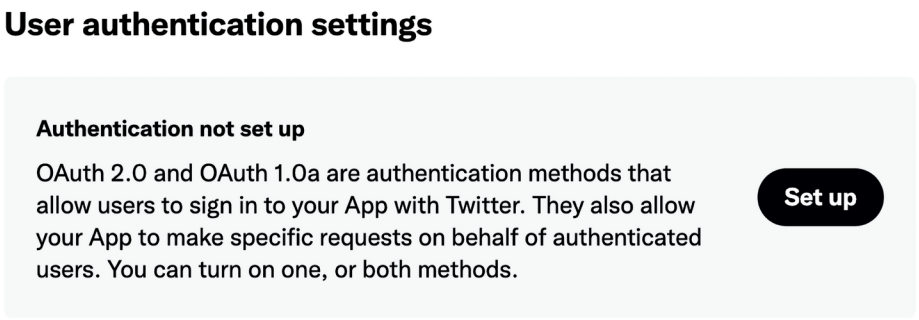


Figure 5.28: Twitter application user authentication setup

Under **User authentication settings**, click on **Set up**. You will see the following screen:

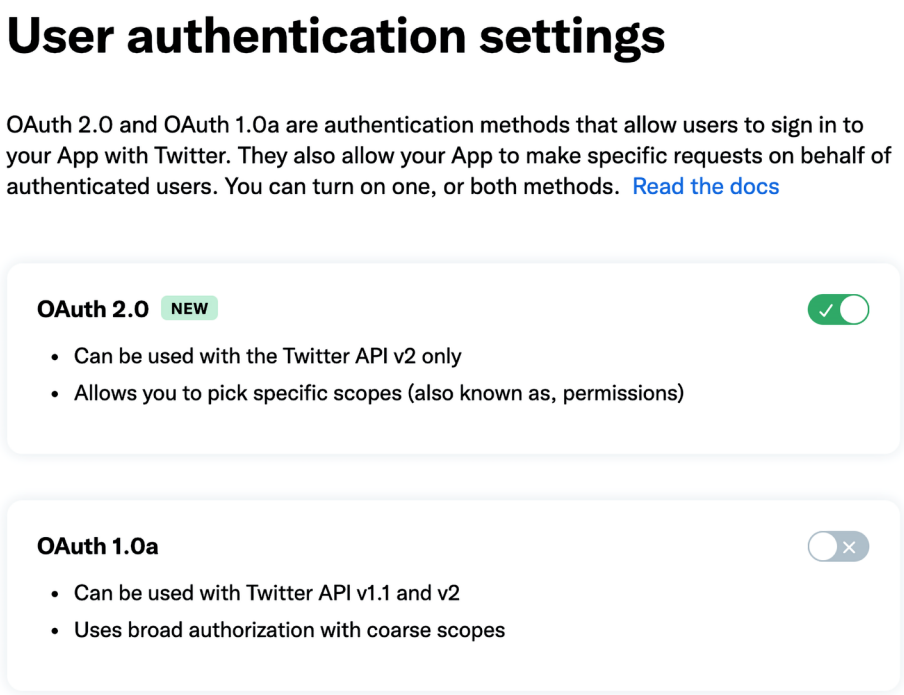


Figure 5.29: Twitter application OAuth 2.0 activation

Activate the **OAuth 2.0** option. This is the OAuth version that we will use. Then, under **OAuth 2.0 Settings**, select **Web App** for **Type of App** as follows:

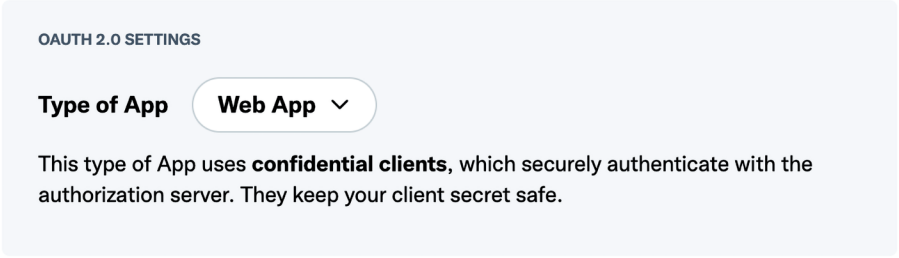


Figure 5.30: Twitter application OAuth 2.0 settings

Under **General Authentication Settings**, enter the following details of your application:

- **Callback URI / Redirect URL:** `https://mysite.com:8000/social-auth/complete/twitter/`
- **Website URL:** `https://mysite.com:8000/`

The settings should look as follows:

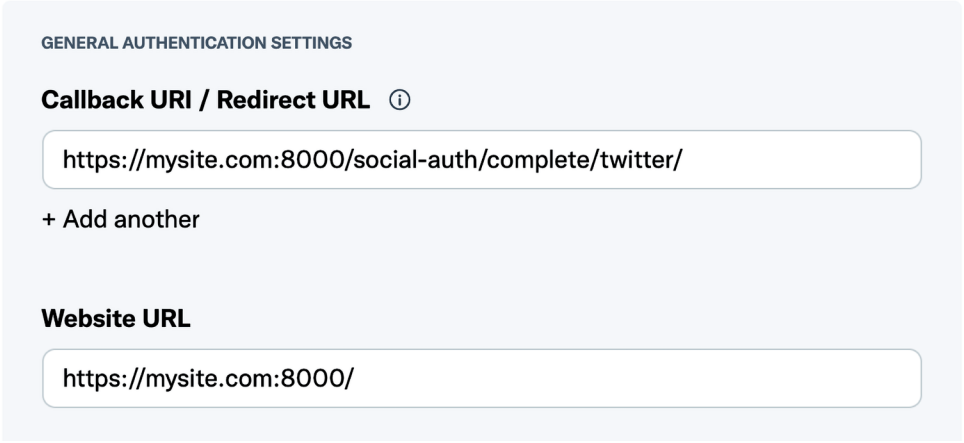


Figure 5.31: Twitter authentication URL configuration

Click on **Save**. Now, you will see the following screen including the **Client ID** and **Client Secret**:

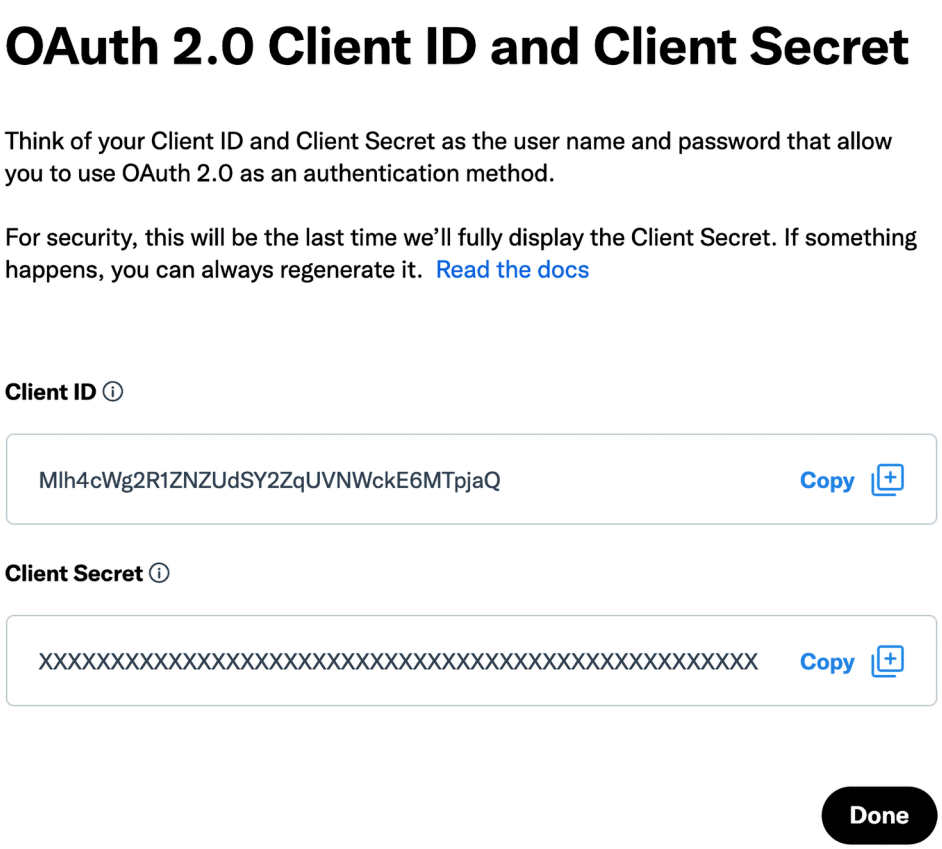


Figure 5.32: Twitter application Client ID and Client Secret

You won't need them for client authentication because you will be using the **API Key** and **API Key Secret** instead. However, you can copy them and store the **Client Secret** in a safe place. Click on **Done**.

You will see another reminder to save the **Client Secret**:

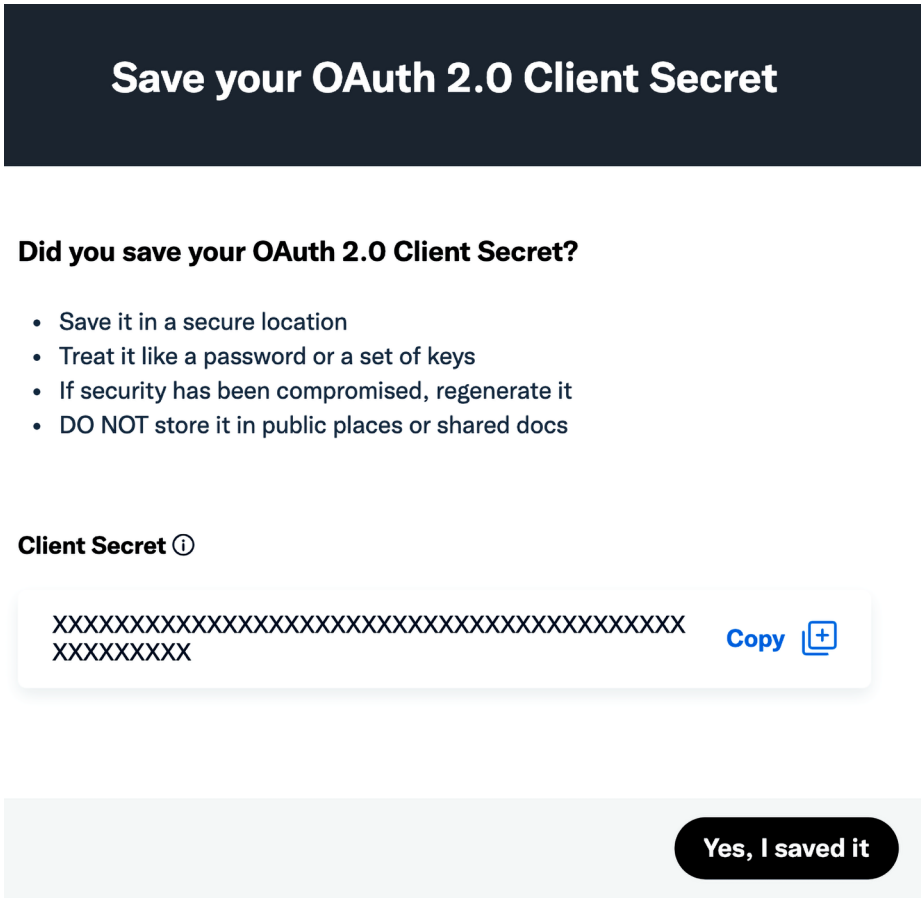


Figure 5.33: Twitter Client Secret reminder

Click on **Yes, I saved it**. Now you will see that OAuth 2.0 authentication has been turned on like in the following screen:

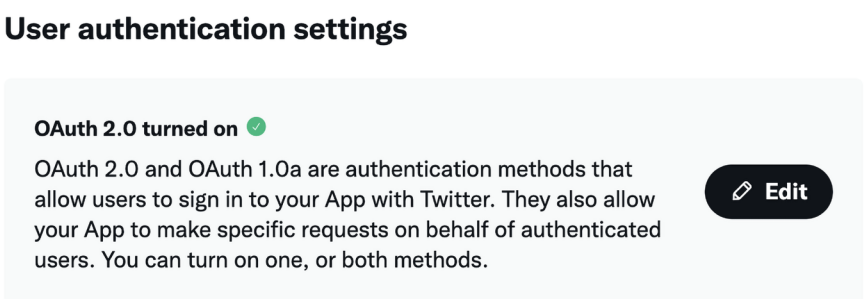


Figure 5.34: Twitter application authentication settings

Now edit the `registration/login.html` template and add the following code highlighted in bold to the `<ul>` element:

```
<ul>
  <li class="facebook">
    <a href="{% url "social:begin" "facebook" %}">
      Sign in with Facebook
    </a>
  </li>
  <li class="twitter">
    <a href="{% url "social:begin" "twitter" %}">
      Sign in with Twitter
    </a>
  </li>
</ul>
```

Use the management command `runserver_plus` provided by Django Extensions to run the development server, as follows:

```
python manage.py runserver_plus --cert-file cert.crt
```

Open `https://mysite.com:8000/account/login/` in your browser. Now, the login page will look as follows:

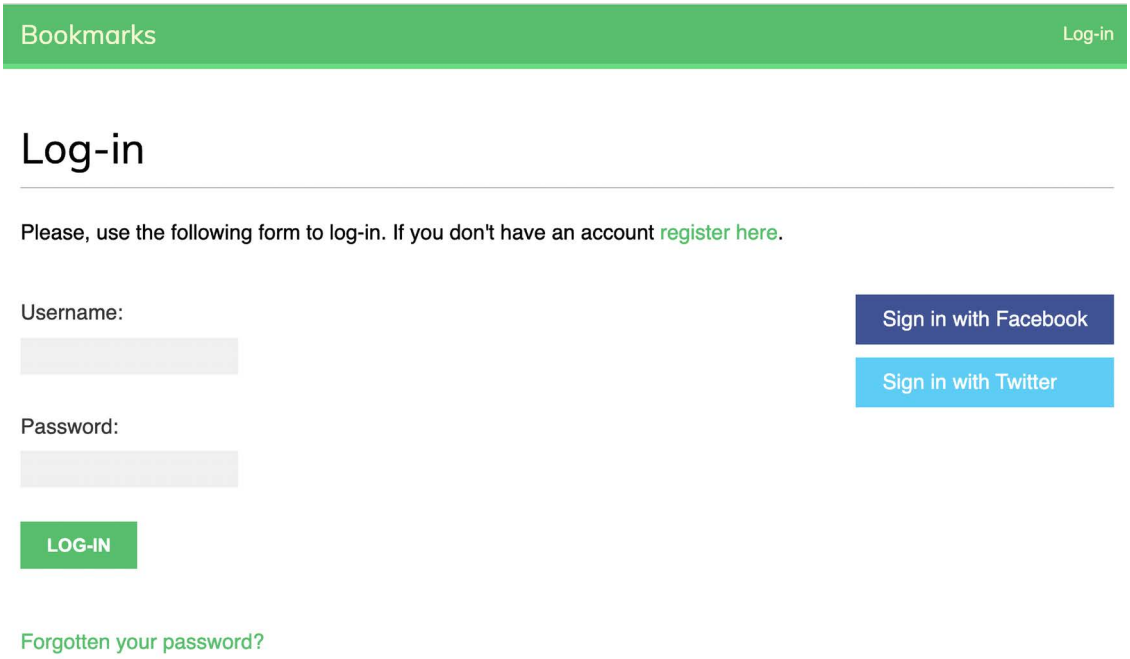


Figure 5.35: The login page including the button for Twitter authentication

Click on the **Sign in with Twitter** link. You will be redirected to Twitter, where you will be asked to authorize the application as follows:

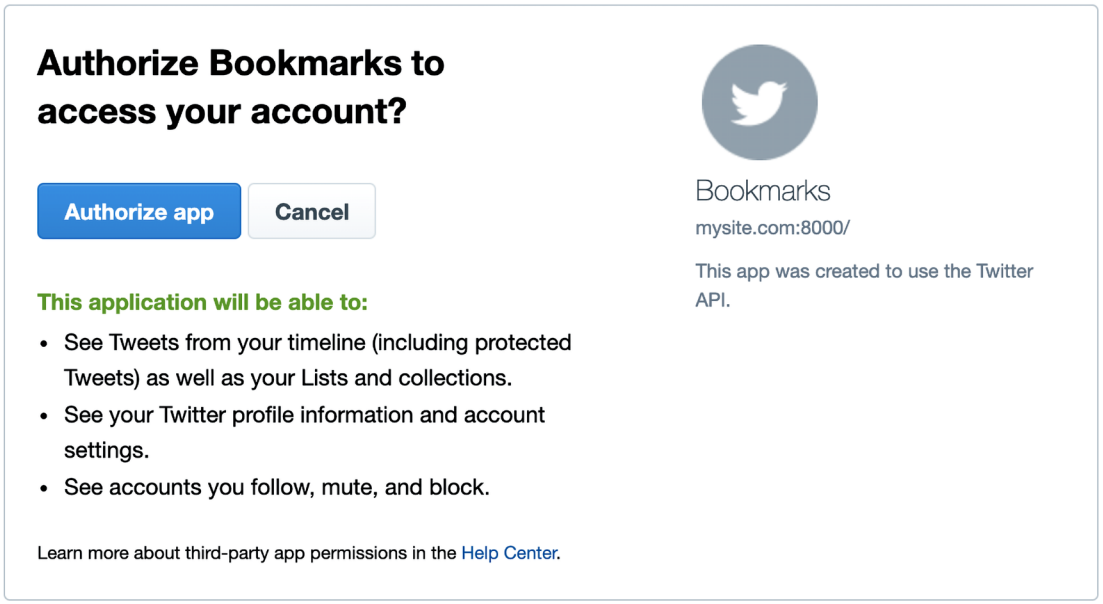


Figure 5.36: Twitter user authorization screen

Click on **Authorize app**. You will briefly see the following page while you are redirected to the dashboard page:

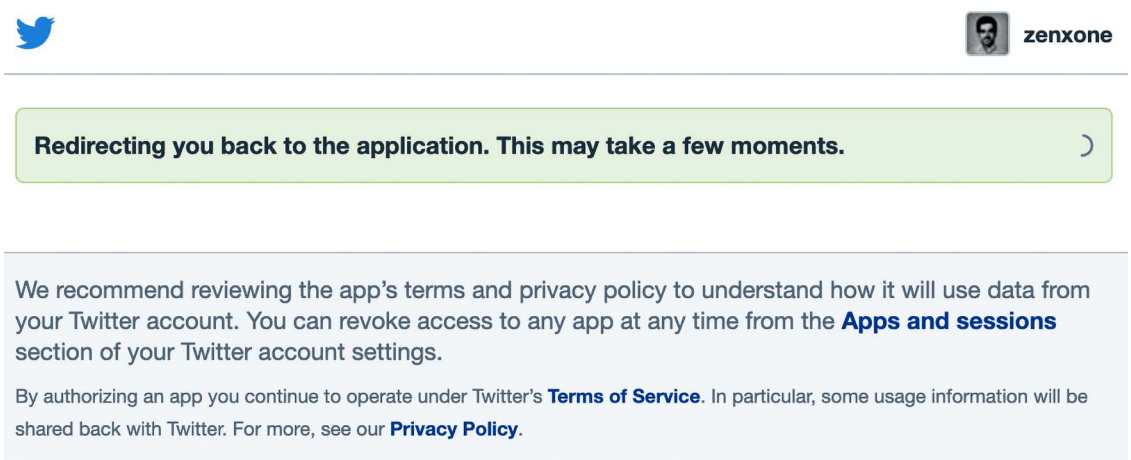


Figure 5.37: Twitter user authentication redirect page

You will then be redirected to the dashboard page of your application.

Authentication using Google

Google offers social authentication using OAuth2. You can read about Google's OAuth2 implementation at `https://developers.google.com/identity/protocols/OAuth2`.

To implement authentication using Google, add the following line highlighted in bold to the `AUTHENTICATION_BACKENDS` setting in the `settings.py` file of your project:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
    'social_core.backends.facebook.FacebookOAuth2',
    'social_core.backends.twitter.TwitterOAuth',
    'social_core.backends.google.GoogleOAuth2',
]
```

First, you will need to create an API key in your Google Developer Console. Open `https://console.cloud.google.com/projectcreate` in your browser. You will see the following screen:

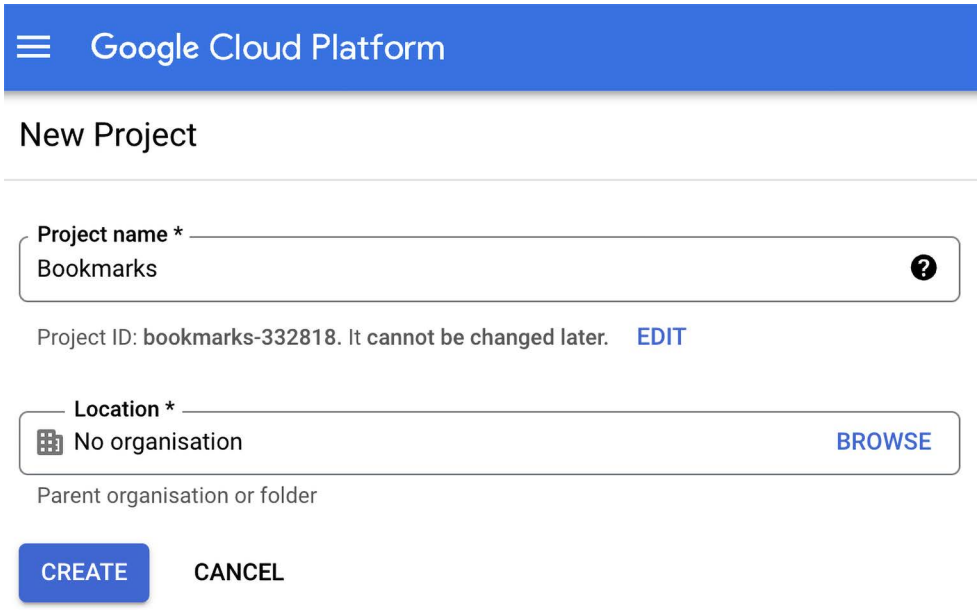


Figure 5.38: The Google project creation form

Under **Project name** enter `Bookmarks` and click the **CREATE** button.

When the new project is ready, make sure the project is selected in the top navigation bar as follows:

Figure 5.39: The Google Developer Console top navigation bar

After the project is created, under **APIs and services**, click on **Credentials** as follows:

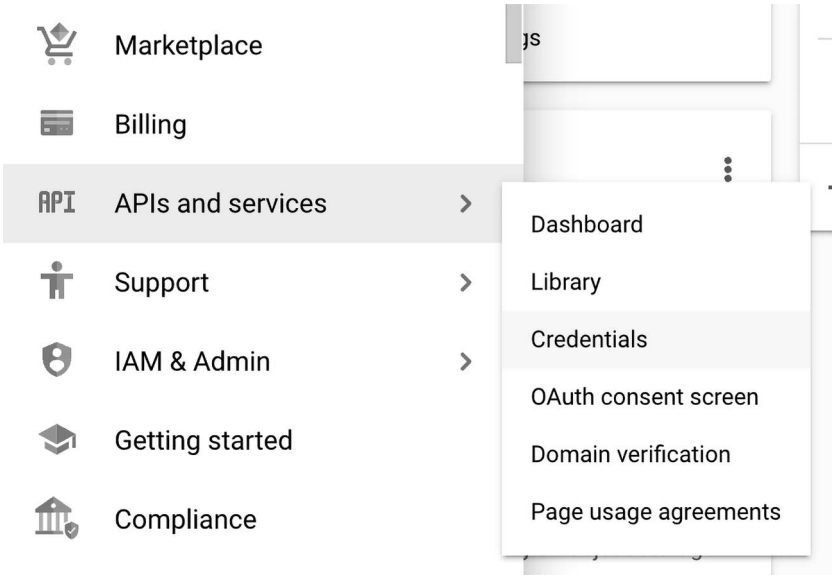


Figure 5.40: Google APIs and services menu

You will see the following screen:

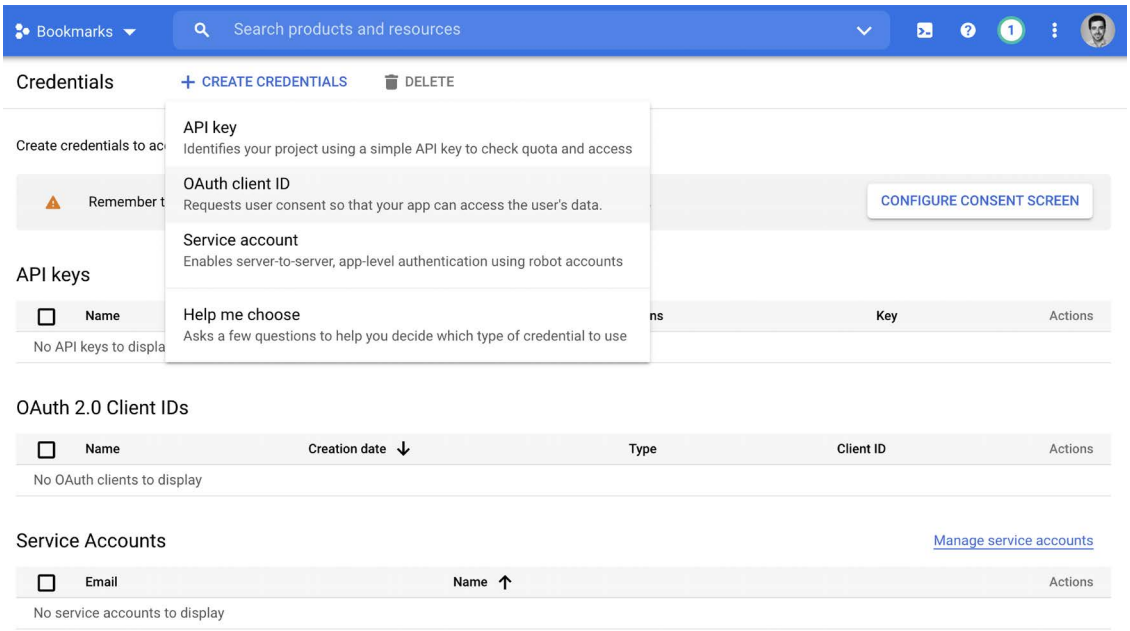


Figure 5.41: Google API creation of API credentials

Then click on **CREATE CREDENTIALS** and click on **OAuth client ID**.

Google will ask you to configure the consent screen first, like this:

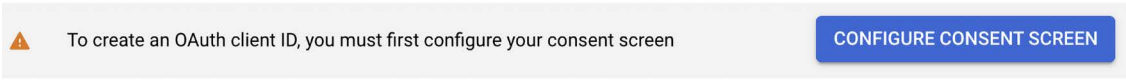


Figure 5.42: The alert to configure the OAuth consent screen

We will configure the page that will be shown to users to give their consent to access your site with their Google account. Click on the **CONFIGURE CONSENT SCREEN** button. You will be redirected to the following screen:

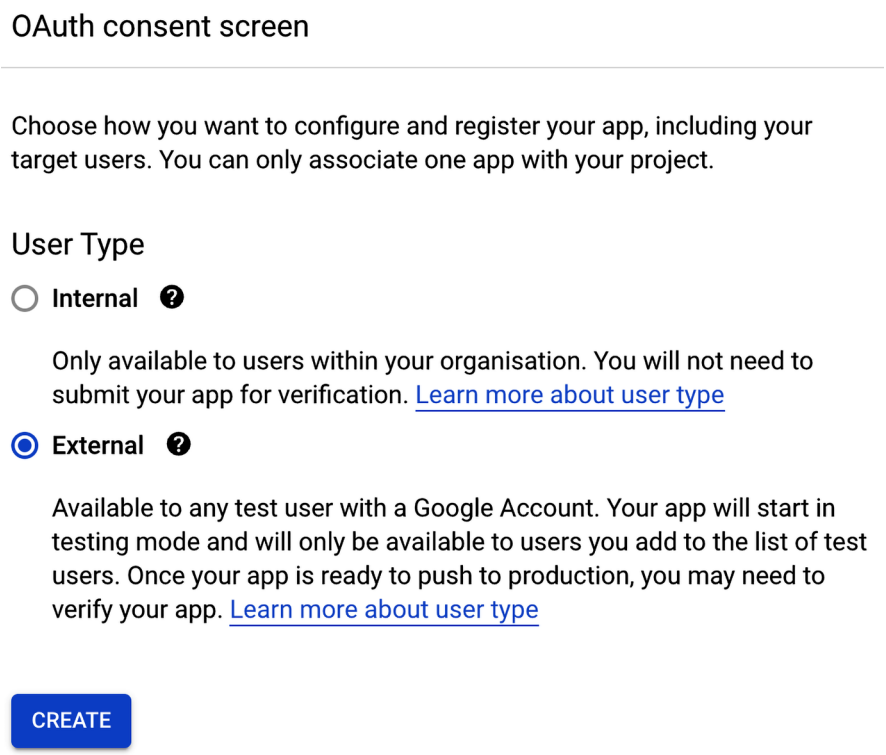


Figure 5.43: User type selection in the Google OAuth consent screen setup

Choose **External** for **User Type** and click the **CREATE** button. You will see the following screen:

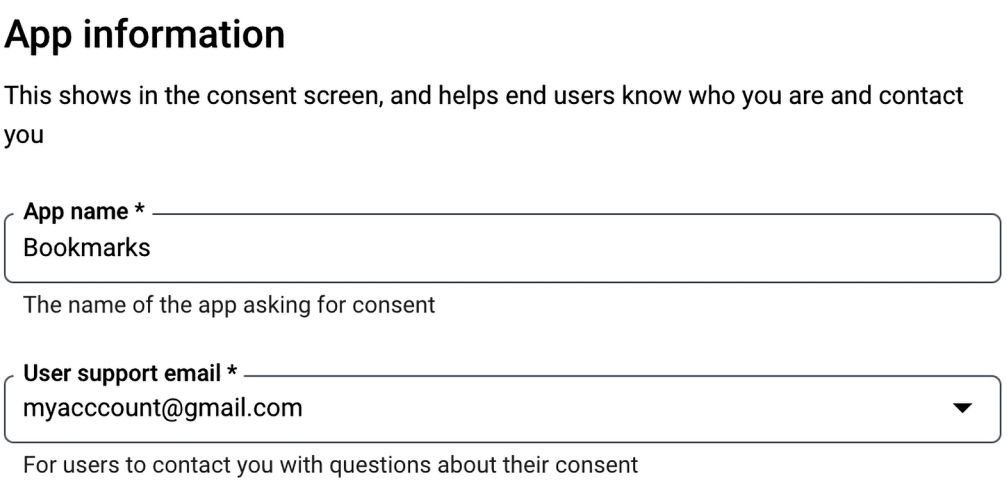


Figure 5.44: Google OAuth consent screen setup

Under **App name**, enter `Bookmarks` and select your email for **User support email**.

Under **Authorised domains**, enter `mysite.com` as follows:

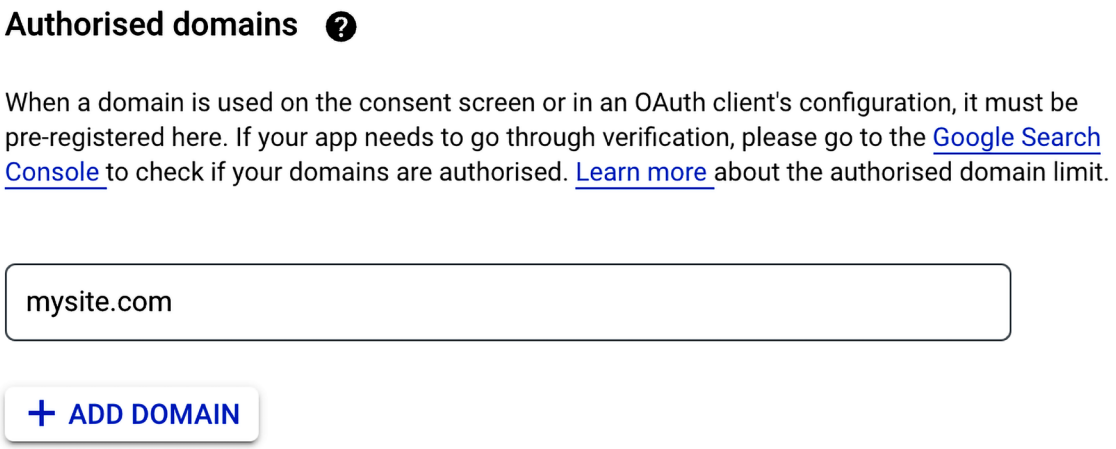


Figure 5.45: Google OAuth authorized domains

Enter your email under **Developer contact information** and click on **SAVE AND CONTINUE**.

In step **2. Scopes**, don't change anything and click on **SAVE AND CONTINUE**.

In step **3. Test users,** add your Google user to **Test users** and click on **SAVE AND CONTINUE** as follows:

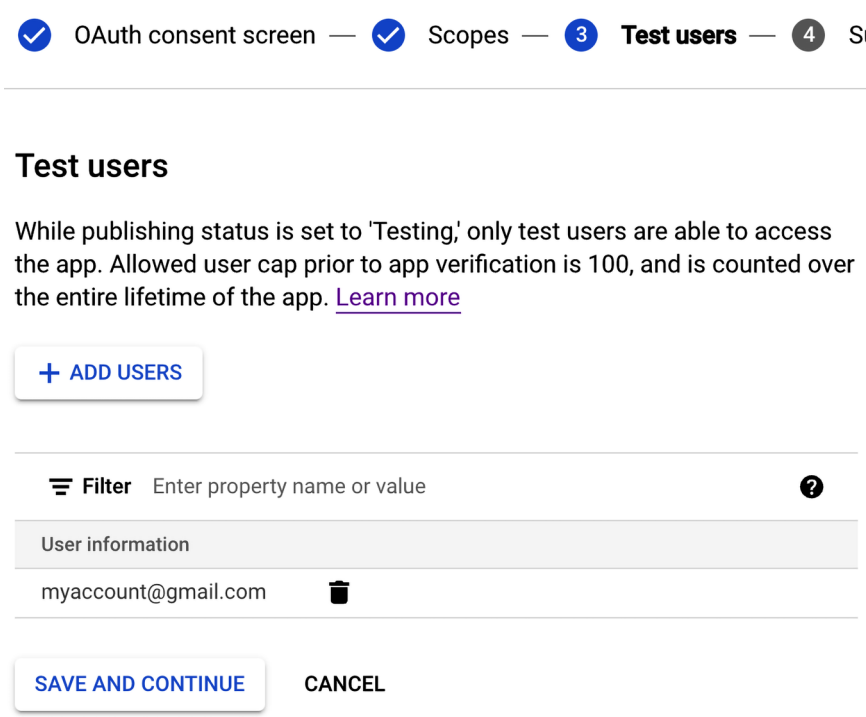


Figure 5.46: Google OAuth test users

You will see a summary of your consent screen configuration. Click on **Back to dashboard.**

In the menu on the left sidebar, click on **Credentials** and click again on **Create credentials** and then on **OAuth client ID.**

As the next step, enter the following information:

- **Application type:** Select **Web application**
- **Name:** Enter `Bookmarks`
- **Authorised JavaScript origins:** Add `https://mysite.com:8000/`
- **Authorised redirect URIs:** Add `https://mysite.com:8000/social-auth/complete/google-oauth2/`

The form should look like this:

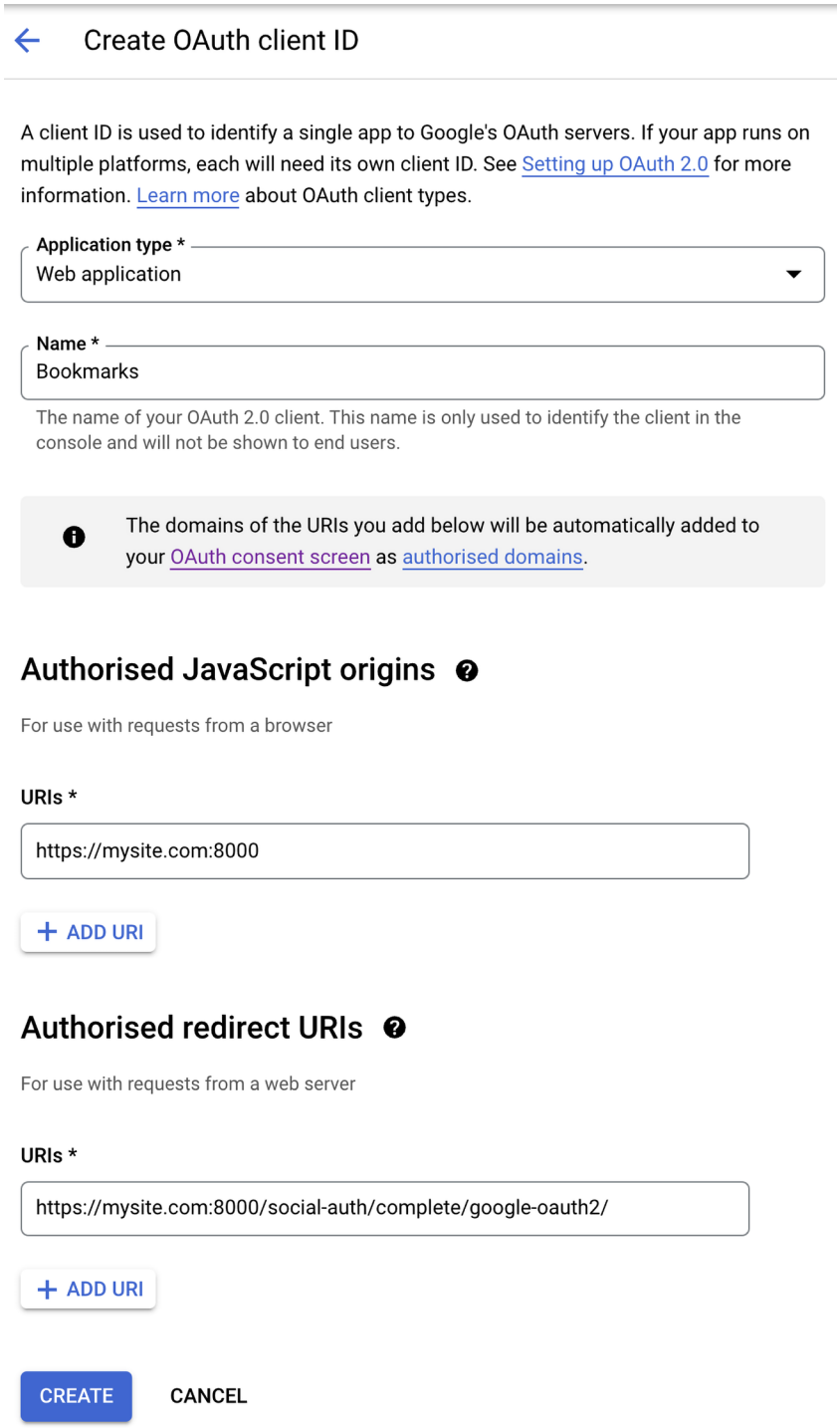


Figure 5.47: The Google OAuth client ID creation form

Click the **CREATE** button. You will get **Your Client ID** and **Your Client Secret** keys:

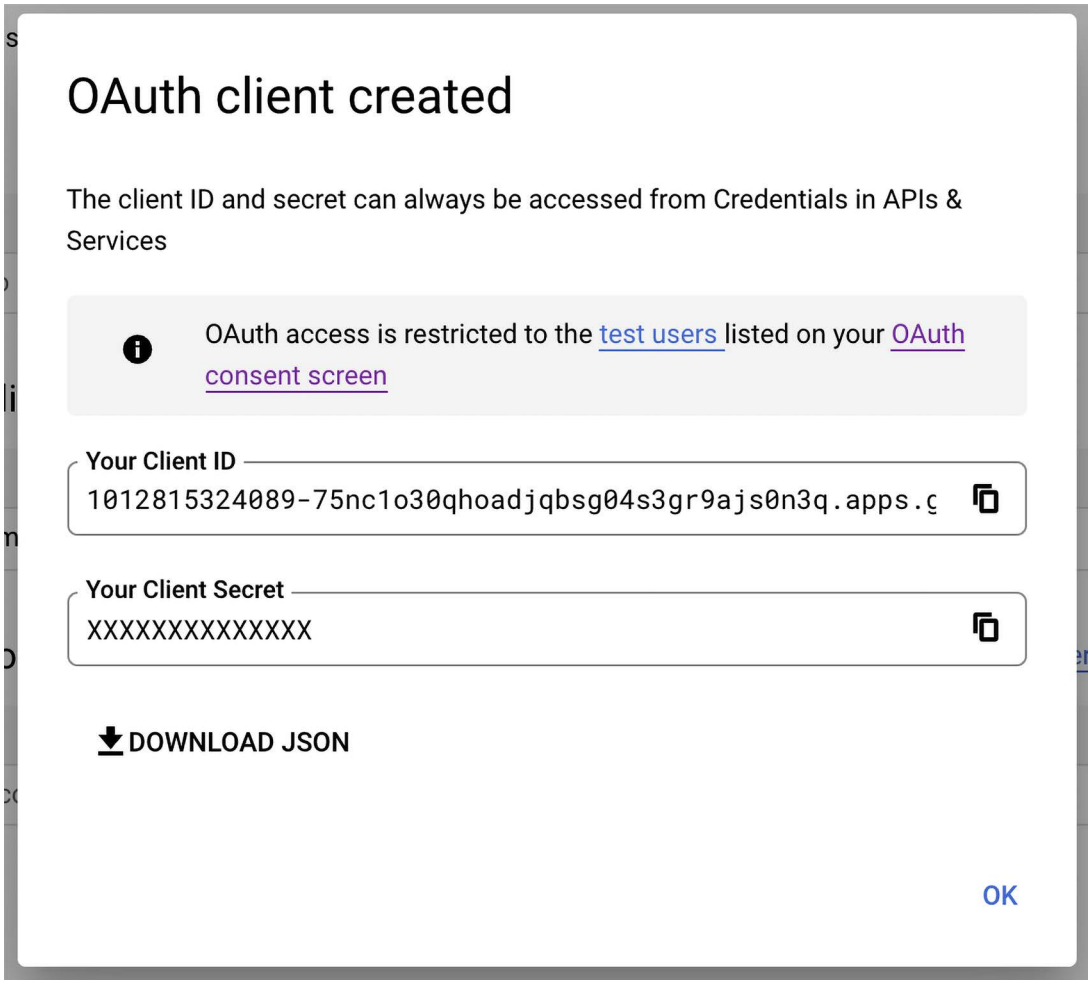


Figure 5.48: Google OAuth Client ID and Client Secret

Add both keys to your `settings.py` file, like this:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'XXX' # Google Client ID
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'XXX' # Google Client Secret
```

Edit the `registration/login.html` template and add the following code highlighted in bold to the `<ul>` element:

```
<ul>
  <li class="facebook">
    <a href="{% url "social:begin" "facebook" %}">
      Sign in with Facebook
    </a>
```

```
    </li>
    <li class="twitter">
      <a href="{% url "social:begin" "twitter" %}">
        Sign in with Twitter
      </a>
    </li>
    <li class="google">
      <a href="{% url "social:begin" "google-oauth2" %}">
        Sign in with Google
      </a>
    </li>
</ul>
```

Use the management command `runserver_plus` provided by Django Extensions to run the development server, as follows:

```
python manage.py runserver_plus --cert-file cert.crt
```

Open `https://mysite.com:8000/account/login/` in your browser. The login page should now look as follows:

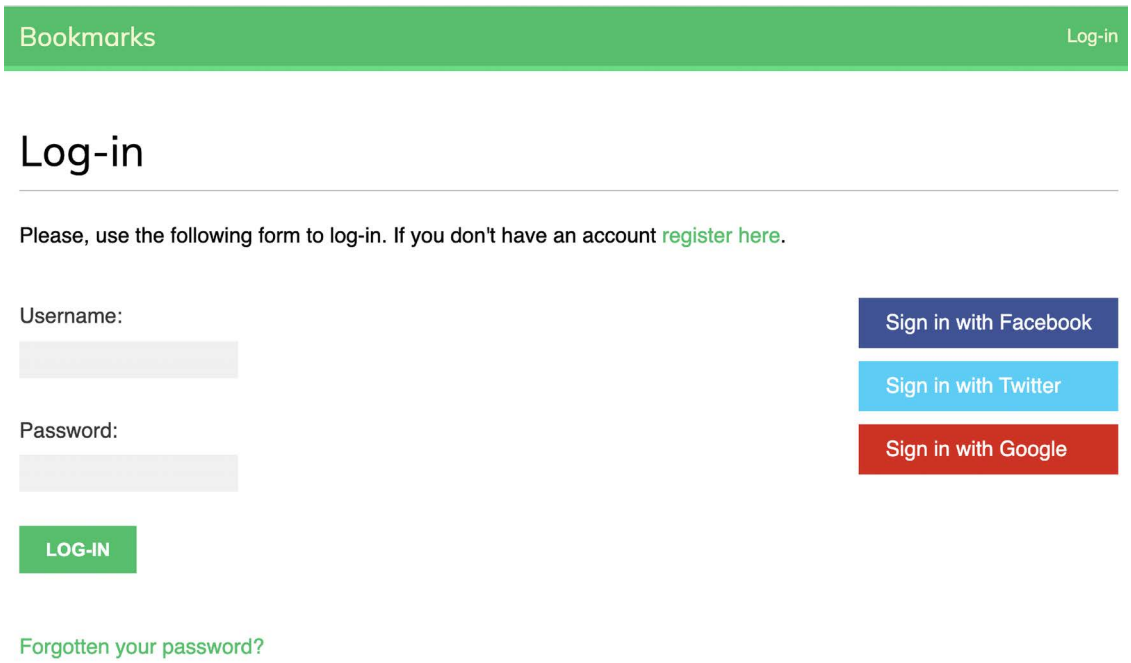


Figure 5.49: The login page including buttons for Facebook, Twitter, and Google authentication

Click on the **Sign in with Google** button. You will see the following screen:

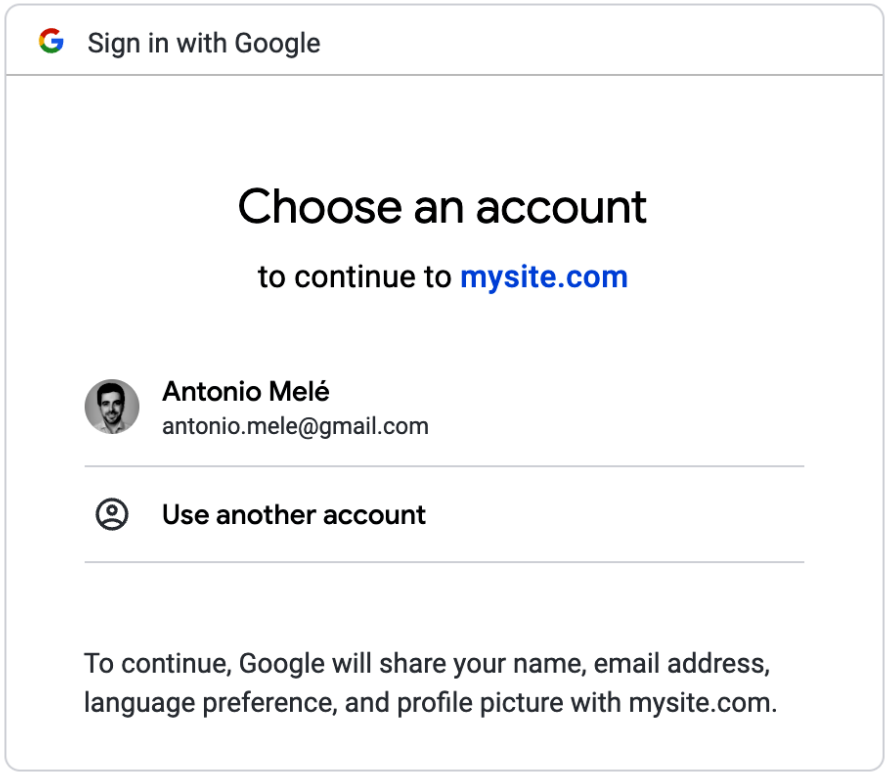


Figure 5.50: The Google application authorization screen

Click on your Google account to authorize the application. You will be logged in and redirected to the dashboard page of your website.

You have now added social authentication to your project with some of the most popular social platforms. You can easily implement social authentication with other online services using Python Social Auth.

Creating a profile for users that register with social authentication

When a user authenticates using social authentication, a new `User` object is created if there isn't an existing user associated with that social profile. Python Social Auth uses a pipeline consisting of a set of functions that are executed in a specific order executed during the authentication flow. These functions take care of retrieving any user details, creating a social profile in the database, and associating it to an existing user or creating a new one.

Currently, a no `Profile` object is created when new users are created via social authentication. We will add a new step to the pipeline, to automatically create a `Profile` object in the database when a new user is created.

Add the following `SOCIAL_AUTH_PIPELINE` setting to the `settings.py` file of your project:

```
SOCIAL_AUTH_PIPELINE = [
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.user.create_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
]
```

This is the default authentication pipeline used by Python Social Auth. It consists of several functions that perform different tasks when authenticating a user. You can find more details about the default authentication pipeline at `https://python-social-auth.readthedocs.io/en/latest/pipeline.html`.

Let's build a function that creates a `Profile` object in the database whenever a new user is created. We will then add this function to the social authentication pipeline.

Edit the `account/authentication.py` file and add the following code to it:

```
from account.models import Profile

def create_profile(backend, user, *args, **kwargs):
    """
    Create user profile for social authentication
    """
    Profile.objects.get_or_create(user=user)
```

The `create_profile` function takes two required arguments:

- `backend`: The social auth backend used for the user authentication. Remember you added the social authentication backends to the `AUTHENTICATION_BACKENDS` setting in your project.
- `user`: The `User` instance of the new or existing user authenticated.

You can check the different arguments that are passed to the pipeline functions at `https://python-social-auth.readthedocs.io/en/latest/pipeline.html#extending-the-pipeline`.

In the `create_profile` function, we check that a `user` object is present and we use the `get_or_create()` method to look up a `Profile` object for the given user, creating one if necessary.

Now, we need to add the new function to the authentication pipeline. Add the following line highlighted in bold to the `SOCIAL_AUTH_PIPELINE` setting in your `settings.py` file:

```
SOCIAL_AUTH_PIPELINE = [
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.user.create_user',
    'account.authentication.create_profile',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
]
```

We have added the `create_profile` function after `social_core.pipeline.create_user`. At this point, a `User` instance is available. The user can be an existing user or a new one created in this step of the pipeline. The `create_profile` function uses the `User` instance to look up the related `Profile` object and create a new one if necessary.

Access the user list in the administration site at `https://mysite.com:8000/admin/auth/user/`. Remove any users created through social authentication.

Then open `https://mysite.com:8000/account/login/` and perform social authentication for the user you deleted. A new user will be created and now a `Profile` object will be created as well. Access `https://mysite.com:8000/admin/account/profile/` to verify that a profile has been created for the new user.

We have successfully added the functionality to create the user profile automatically for social authentication.

Python Social Auth also offers a pipeline mechanism for the disconnection flow. You can find more details at `https://python-social-auth.readthedocs.io/en/latest/pipeline.html#disconnection-pipeline`.

Additional resources

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

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter05`
- Python Social Auth – `https://github.com/python-social-auth`
- Python Social Auth's authentication backends – `https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends`

- Django allowed hosts setting – `https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts`
- Django Extensions documentation – `https://django-extensions.readthedocs.io/en/latest/`
- Facebook developer portal – `https://developers.facebook.com/apps/`
- Twitter apps – `https://developer.twitter.com/en/apps/create`
- Google's OAuth2 implementation – `https://developers.google.com/identity/protocols/OAuth2`
- Google APIs credentials – `https://console.developers.google.com/apis/credentials`
- Python Social Auth pipeline – `https://python-social-auth.readthedocs.io/en/latest/pipeline.html`
- Extending the Python Social Auth pipeline – `https://python-social-auth.readthedocs.io/en/latest/pipeline.html#extending-the-pipeline`
- Python Social Auth pipeline for disconnection – `https://python-social-auth.readthedocs.io/en/latest/pipeline.html#disconnection-pipeline`

Summary

In this chapter, you added social authentication to your site so that users can use their existing Facebook, Twitter, or Google accounts to log in. You used Python Social Auth and implemented social authentication using OAuth 2.0, the industry-standard protocol for authorization. You also learned how to serve your development server through HTTPS using Django Extensions. Finally, you customized the authentication pipeline to create user profiles for new users automatically.

In the next chapter, you will create an image bookmarking system. You will create models with many-to-many relationships and customize the behavior of forms. You will learn how to generate image thumbnails and how to build AJAX functionalities using JavaScript and Django.

6

Sharing Content on Your Website

In the previous chapter, you used Django Social Auth to add social authentication to your site using Facebook, Google, and Twitter. You learned how to run your development server with HTTPS on your local machine using Django Extensions. You customized the social authentication pipeline to create a user profile for new users automatically.

In this chapter, you will learn how to create a JavaScript bookmarklet to share content from other sites on your website, and you will implement AJAX features in your project using JavaScript and Django.

This chapter will cover the following points:

- Creating many-to-many relationships
- Customizing behavior for forms
- Using JavaScript with Django
- Building a JavaScript bookmarklet
- Generating image thumbnails using `easy-thumbnails`
- Implementing asynchronous HTTP requests with JavaScript and Django
- Building infinite scroll pagination

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter06`.

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

Creating an image bookmarking website

We will now learn how to allow users to bookmark images that they find on other websites and share them on our site. To build this functionality, we will need the following elements:

1. A data model to store images and related information
2. A form and a view to handle image uploads

3. JavaScript bookmarklet code that can be executed on any website. This code will find images across the page and allow users to select the image they want to bookmark

First, create a new application inside your `bookmarks` project directory by running the following command in the shell prompt:

```
django-admin startapp images
```

Add the new application to the `INSTALLED_APPS` setting in the `settings.py` file of the project, as follows:

```
INSTALLED_APPS = [
    # ...
    'images.apps.ImagesConfig',
]
```

We have activated the `images` application in the project.

Building the image model

Edit the `models.py` file of the `images` application and add the following code to it:

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

class Image(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
                             related_name='images_created',
                             on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200,
                            blank=True)
    url = models.URLField(max_length=2000)
    image = models.ImageField(upload_to='images/%Y/%m/%d/')
    description = models.TextField(blank=True)
    created = models.DateField(auto_now_add=True)

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

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

This is the model that we will use to store images in the platform. Let's take a look at the fields of this model:

- `user`: This indicates the `User` object that bookmarked this image. This is a foreign key field because it specifies a one-to-many relationship: a user can post multiple images, but each image is posted by a single user. We have used `CASCADE` for the `on_delete` parameter so that related images are deleted when a user is deleted.
- `title`: A title for the image.
- `slug`: A short label that contains only letters, numbers, underscores, or hyphens to be used for building beautiful SEO-friendly URLs.
- `url`: The original URL for this image. We use `max_length` to define a maximum length of `2000` characters.
- `image`: The image file.
- `description`: An optional description for the image.
- `created`: The date and time that indicate when the object was created in the database. We have added `auto_now_add` to automatically set the current datetime when the object is created.

In the `Meta` class of the model, we have defined a database index in descending order for the `created` field. We have also added the `ordering` attribute to tell Django that it should sort results by the `created` field by default. We indicate descending order by using a hyphen before the field name, such as `-created`, so that new images will be displayed first.

Database indexes improve query performance. Consider creating indexes for fields that you frequently query using `filter()`, `exclude()`, or `order_by()`. `ForeignKey` fields or fields with `unique=True` imply the creation of an index. You can learn more about database indexes at `https://docs.djangoproject.com/en/4.1/ref/models/options/#django.db.models.Options.indexes`.

We will override the `save()` method of the `Image` model to automatically generate the `slug` field based on the value of the `title` field. Import the `slugify()` function and add a `save()` method to the `Image` model, as follows. New lines are highlighted in bold:

```
from django.utils.text import slugify

class Image(models.Model):
    # ...
    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super().save(*args, **kwargs)
```

When an `Image` object is saved, if the `slug` field doesn't have a value, the `slugify()` function is used to automatically generate a slug from the `title` field of the image. The object is then saved. By generating slugs automatically from the title, users won't have to provide a slug when they share images on our website.

Creating many-to-many relationships

Next, we will add another field to the `Image` model to store the users who like an image. We will need a many-to-many relationship in this case because a user might like multiple images and each image can be liked by multiple users.

Add the following field to the `Image` model:

```
users_like = models.ManyToManyField(settings.AUTH_USER_MODEL,
                                    related_name='images_liked',
                                    blank=True)
```

When we define a `ManyToManyField` field, Django creates an intermediary join table using the primary keys of both models. *Figure 6.1* shows the database table that will be created for this relationship:

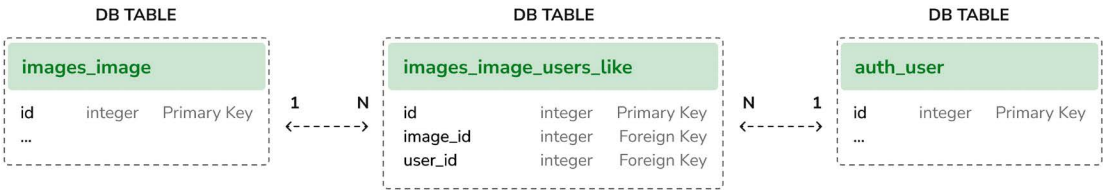


Figure 6.1: Intermediary database table for the many-to-many relationship

The `images_image_users_like` table is created by Django as an intermediary table that has references to the `images_image` table (`Image` model) and `auth_user` table (`User` model). The `ManyToManyField` field can be defined in either of the two related models.

As with `ForeignKey` fields, the `related_name` attribute of `ManyToManyField` allows you to name the relationship from the related object back to this one. `ManyToManyField` fields provide a many-to-many manager that allows you to retrieve related objects, such as `image.users_like.all()`, or get them from a `user` object, such as `user.images_liked.all()`.

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

Open the shell prompt and run the following command to create an initial migration:

```
python manage.py makemigrations images
```

The output should be similar to the following one:

```
Migrations for 'images':
  images/migrations/0001_initial.py
    - Create model Image
```

```
    - Create index images_imag_created_d57897_idx on field(s) -created of model
image
```

Now run the following command to apply your migration:

```
python manage.py migrate images
```

You will get an output that includes the following line:

```
Applying images.0001_initial... OK
```

The `Image` model is now synced to the database.

Registering the image model in the administration site

Edit the `admin.py` file of the `images` application and register the `Image` model into the administration site, as follows:

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

@admin.register(Image)
class ImageAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'image', 'created']
    list_filter = ['created']
```

Start the development server with the following command:

```
python manage.py runserver_plus --cert-file cert.crt
```

Open `https://127.0.0.1:8000/admin/` in your browser, and you will see the `Image` model in the administration site, like this:

Figure 6.2: The Images block on the Django administration site index page

You have completed the model to store images. Now you will learn how to implement a form to retrieve images by their URL and store them using the `Image` model.

Posting content from other websites

We will allow users to bookmark images from external websites and share them on our site. Users will provide the URL of the image, a title, and an optional description. We will create a form and a view to download the image and create a new `Image` object in the database.

Let's start by building a form to submit new images.

Create a new `forms.py` file inside the `images` application directory and add the following code to it:

```
from django import forms
from .models import Image

class ImageCreateForm(forms.ModelForm):
    class Meta:
        model = Image
        fields = ['title', 'url', 'description']
        widgets = {
            'url': forms.HiddenInput,
        }
```

We have defined a `ModelForm` form from the `Image` model, including only the `title`, `url`, and `description` fields. Users will not enter the image URL directly in the form. Instead, we will provide them with a JavaScript tool to choose an image from an external site, and the form will receive the image's URL as a parameter. We have overridden the default widget of the `url` field to use a `HiddenInput` widget. This widget is rendered as an HTML `input` element with a `type="hidden"` attribute. We use this widget because we don't want this field to be visible to users.

Cleaning form fields

In order to verify that the provided image URL is valid, we will check that the filename ends with a `.jpg`, `.jpeg`, or `.png` extension to allow sharing JPEG and PNG files only. In the previous chapter, we used the `clean_<fieldname>()` convention to implement field validation. This method is executed for each field, if present, when we call `is_valid()` on a form instance. In the `clean` method, you can alter the field's value or raise any validation errors for the field.

In the `forms.py` file of the `images` application, add the following method to the `ImageCreateForm` class:

```
def clean_url(self):
    url = self.cleaned_data['url']
    valid_extensions = ['jpg', 'jpeg', 'png']
    extension = url.rsplit('.', 1)[1].lower()
    if extension not in valid_extensions:
        raise forms.ValidationError('The given URL does not ' \
                                    'match valid image extensions.')
    return url
```

In the preceding code, we have defined a `clean_url()` method to clean the `url` field. The code works as follows:

1. The value of the `url` field is retrieved by accessing the `cleaned_data` dictionary of the form instance.
2. The URL is split to check whether the file has a valid extension. If the extension is invalid, a `ValidationError` is raised, and the form instance is not validated.

In addition to validating the given URL, we also need to download the image file and save it. We could, for example, use the view that handles the form to download the image file. Instead, let's take a more general approach by overriding the `save()` method of the model form to perform this task when the form is saved.

Installing the Requests library

When a user bookmarks an image, we will need to download the image file by its URL. We will use the Requests Python library for this purpose. Requests is the most popular HTTP library for Python. It abstracts the complexity of dealing with HTTP requests and provides a very simple interface to consume HTTP services. You can find the documentation for the Requests library at `https://requests.readthedocs.io/en/master/`.

Open the shell and install the Requests library with the following command:

```
pip install requests==2.28.1
```

We will now override the `save()` method of `ImageCreateForm` and use the Requests library to retrieve the image by its URL.

Overriding the save() method of a ModelForm

As you know, `ModelForm` provides a `save()` method to save the current model instance to the database and return the object. This method receives a Boolean `commit` parameter, which allows you to specify whether the object has to be persisted to the database. If `commit` is `False`, the `save()` method will return a model instance but will not save it to the database. We will override the form's `save()` method in order to retrieve the image file by the given URL and save it to the file system.

Add the following imports at the top of the `forms.py` file:

```
from django.core.files.base import ContentFile
from django.utils.text import slugify
import requests
```

Then, add the following `save()` method to the `ImageCreateForm` form:

```
def save(self, force_insert=False,
               force_update=False,
               commit=True):
    image = super().save(commit=False)
    image_url = self.cleaned_data['url']
    name = slugify(image.title)
    extension = image_url.rsplit('.', 1)[1].lower()
    image_name = f'{name}.{extension}'
    # download image from the given URL
    response = requests.get(image_url)
    image.image.save(image_name,
```

```
                    ContentFile(response.content),
                    save=False)
    if commit:
        image.save()
    return image
```

We have overridden the `save()` method, keeping the parameters required by `ModelForm`. The preceding code can be explained as follows:

1. A new `image` instance is created by calling the `save()` method of the form with `commit=False`.
2. The URL of the image is retrieved from the `cleaned_data` dictionary of the form.
3. An image name is generated by combining the `image` title slug with the original file extension of the image.
4. The Requests Python library is used to download the image by sending an HTTP `GET` request using the image URL. The response is stored in the `response` object.
5. The `save()` method of the `image` field is called, passing it a `ContentFile` object that is instantiated with the downloaded file content. In this way, the file is saved to the media directory of the project. The `save=False` parameter is passed to avoid saving the object to the database yet.
6. To maintain the same behavior as the original `save()` method of the model form, the form is only saved to the database if the `commit` parameter is `True`.

We will need a view to create an instance of the form and handle its submission.

Edit the `views.py` file of the `images` application and add the following code to it. New code is highlighted in bold:

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from .forms import ImageCreateForm

@login_required
def image_create(request):
    if request.method == 'POST':
        # form is sent
        form = ImageCreateForm(data=request.POST)
        if form.is_valid():
            # form data is valid
            cd = form.cleaned_data
            new_image = form.save(commit=False)
            # assign current user to the item
            new_image.user = request.user
            new_image.save()
```

```
            messages.success(request,
                             'Image added successfully')
            # redirect to new created item detail view
            return redirect(new_image.get_absolute_url())
    else:
        # build form with data provided by the bookmarklet via GET
        form = ImageCreateForm(data=request.GET)
    return render(request,
                  'images/image/create.html',
                  {'section': 'images',
                   'form': form})
```

In the preceding code, we have created a view to store images on the site. We have added the `login_required` decorator to the `image_create` view to prevent access to unauthenticated users. This is how this view works:

1. Initial data has to be provided through a `GET` HTTP request in order to create an instance of the form. This data will consist of the `url` and `title` attributes of an image from an external website. Both parameters will be set in the `GET` request by the JavaScript bookmarklet that we will create later. For now, we can assume that this data will be available in the request.
2. When the form is submitted with a `POST` HTTP request, it is validated with `form.is_valid()`. If the form data is valid, a new `Image` instance is created by saving the form with `form.save(commit=False)`. The new instance is not saved to the database because of `commit=False`.
3. A relationship to the current user performing the request is added to the new `Image` instance with `new_image.user = request.user`. This is how we will know who uploaded each image.
4. The `Image` object is saved to the database.
5. Finally, a success message is created using the Django messaging framework and the user is redirected to the canonical URL of the new image. We haven't yet implemented the `get_absolute_url()` method of the `Image` model; we will do that later.

Create a new `urls.py` file inside the `images` application and add the following code to it:

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

app_name = 'images'

urlpatterns = [
    path('create/', views.image_create, name='create'),
]
```

Edit the main `urls.py` file of the `bookmarks` project to include the patterns for the `images` application, as follows. The new code is highlighted in bold:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
         include('social_django.urls', namespace='social')),
    path('images/', include('images.urls', namespace='images')),
]
```

Finally, we need to create a template to render the form. Create the following directory structure inside the `images` application directory:

```
templates/
  images/
    image/
      create.html
```

Edit the new `create.html` template and add the following code to it:

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

{% block title %}Bookmark an image{% endblock %}

{% block content %}
  <h1>Bookmark an image</h1>
  <img src="{{ request.GET.url }}" class="image-preview">
  <form method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <input type="submit" value="Bookmark it!">
  </form>
{% endblock %}
```

Run the development server with the following command in the shell prompt:

```
python manage.py runserver_plus --cert-file cert.crt
```

Open `https://127.0.0.1:8000/images/create/?title=...&url=...` in your browser, including the `title` and `url` GET parameters, providing an existing JPEG image URL in the latter. For example, you can use the following URL: `https://127.0.0.1:8000/images/create/?title=%20Django%20and%20Duke&url=https://upload.wikimedia.org/wikipedia/commons/8/85/Django_Reinhardt_and_Duke_Ellington_%28Gottlieb%29.jpg`.

You will see the form with an image preview, like the following:

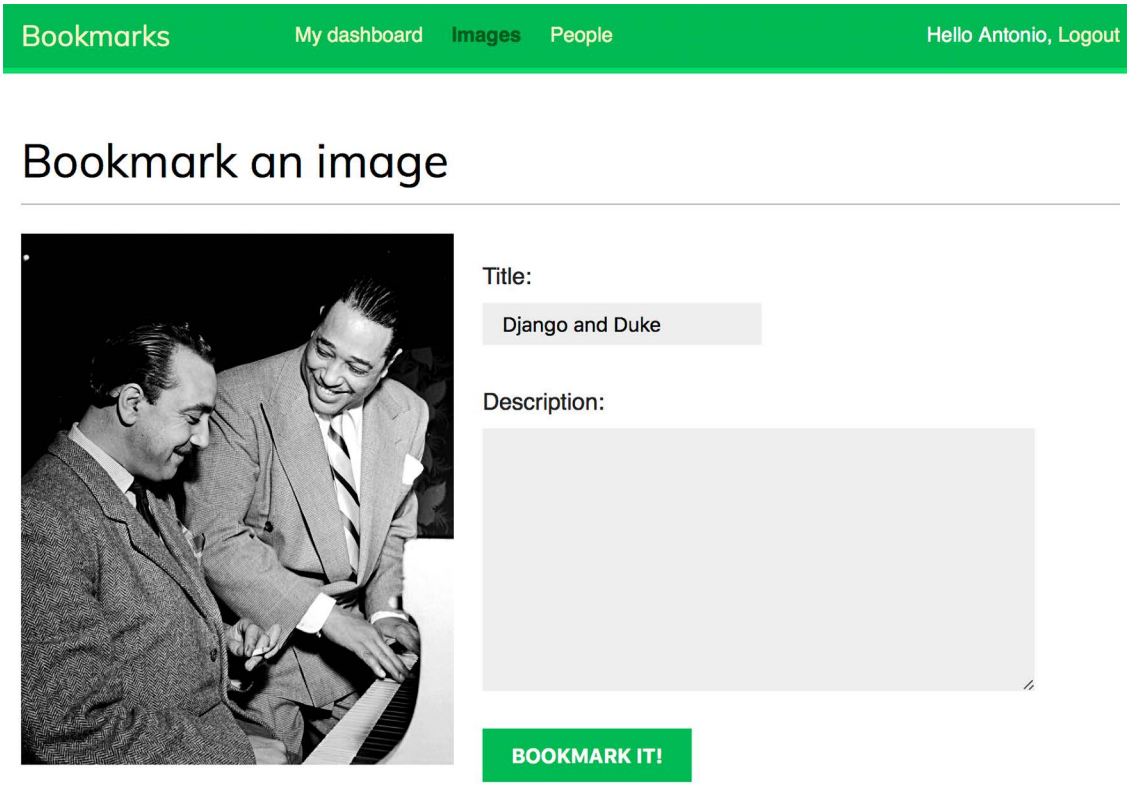


Figure 6.3: The create a new image bookmark page

Add a description and click on the **BOOKMARK IT!** button. A new `Image` object will be saved in your database. However, you will get an error that indicates that the `Image` model has no `get_absolute_url()` method, as follows:

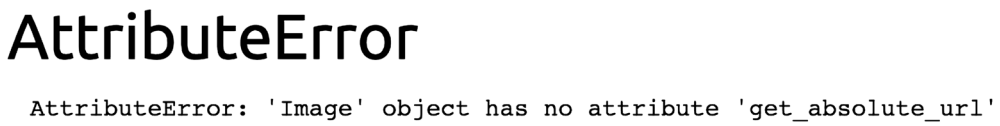


Figure 6.4: An error showing that the Image object has no attribute get_absolute_url

Don't worry about this error for now; we are going to implement the `get_absolute_url` method in the `Image` model later.

Open `https://127.0.0.1:8000/admin/images/image/` in your browser and verify that the new `image` object has been saved, like this:

Figure 6.5: The administration site image list page showing the Image object created

Building a bookmarklet with JavaScript

A bookmarklet is a bookmark stored in a web browser that contains JavaScript code to extend the browser's functionality. When you click on the bookmark in the bookmarks or favorites bar of your browser, the JavaScript code is executed on the website being displayed in the browser. This is very useful for building tools that interact with other websites.

Some online services, such as Pinterest, implement their own bookmarklet to let users share content from other sites onto their platform. The Pinterest bookmarklet, named *browser button*, is available at `https://about.pinterest.com/en/browser-button`. The Pinterest bookmarklet is provided as a Google Chrome extension, a Microsoft Edge add-on, or a plain JavaScript bookmarklet for Safari and other browsers that you can drag and drop to the bookmarks bar of your browser. The bookmarklet allows users to save images or websites to their Pinterest account.

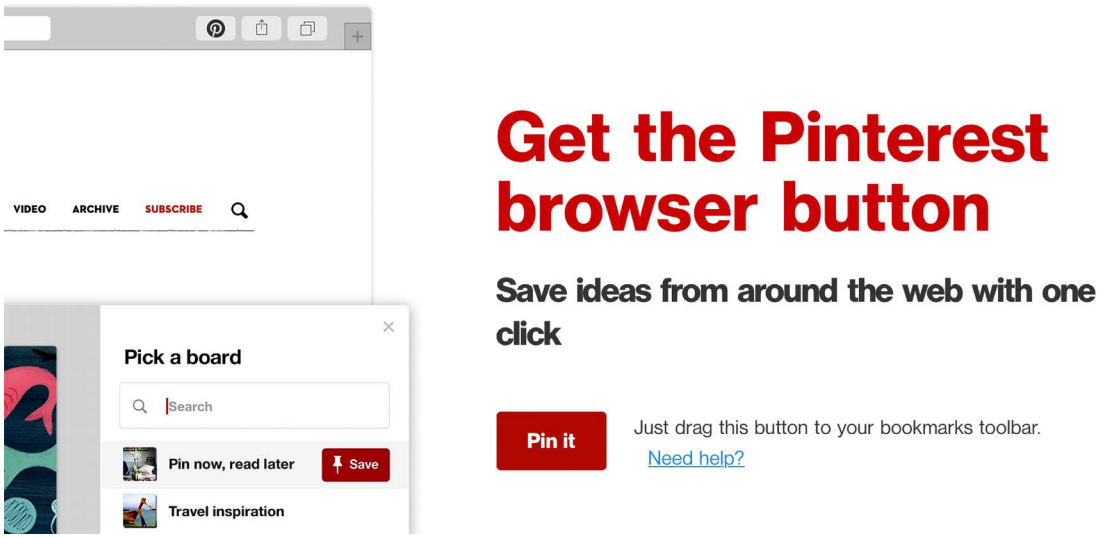


Figure 6.6: The Pin it bookmarklet from Pinterest

Let's create a bookmarklet in a similar way for your website. For that, we will be using JavaScript.

This is how your users will add the bookmarklet to their browser and use it:

1. The user drags a link from your site to their browser's bookmarks bar. The link contains JavaScript code in its `href` attribute. This code will be stored in the bookmark.
2. The user navigates to any website and clicks on the bookmark in the bookmarks or favorites bar. The JavaScript code of the bookmark is executed.

Since the JavaScript code will be stored as a bookmark, we will not be able to update it after the user has added it to their bookmarks bar. This is an important drawback that you can solve by implementing a launcher script. Users will save the launcher script as a bookmark, and the launcher script will load the actual JavaScript bookmarklet from a URL. By doing this, you will be able to update the code of the bookmarklet at any time. This is the approach that we will take to build the bookmarklet. Let's start!

Create a new template under `images/templates/` and name it `bookmarklet_launcher.js`. This will be the launcher script. Add the following JavaScript code to the new file:

```
(function(){
  if(!window.bookmarklet) {
    bookmarklet_js = document.body.appendChild(document.
createElement('script'));
    bookmarklet_js.src = '//127.0.0.1:8000/static/js/bookmarklet.js?r='+Math.
floor(Math.random()*9999999999999999);
    window.bookmarklet = true;
  }
  else {
    bookmarkletLaunch();
  }
})();
```

The preceding script checks whether the bookmarklet has already been loaded by checking the value of the `bookmarklet` window variable with `if(!window.bookmarklet)`:

- If `window.bookmarklet` is not defined or doesn't have a truthy value (considered `true` in a Boolean context), a JavaScript file is loaded by appending a `<script>` element to the body of the HTML document loaded in the browser. The `src` attribute is used to load the URL of the `bookmarklet.js` script with a random 16-digit integer parameter generated with `Math.random()*9999999999999999`. Using a random number, we prevent the browser from loading the file from the browser's cache. If the bookmarklet JavaScript has been previously loaded, the different parameter value will force the browser to load the script from the source URL again. This way, we make sure the bookmarklet always runs the most up-to-date JavaScript code.
- If `window.bookmarklet` is defined and has a truthy value, the function `bookmarkletLaunch()` is executed. We will define `bookmarkletLaunch()` as a global function in the `bookmarklet.js` script.

By checking the `bookmarklet` window variable, we prevent the bookmarklet JavaScript code from being loaded more than once if users click on the bookmarklet repeatedly.

You created the bookmarklet launcher code. The actual bookmarklet code will reside in the `bookmarklet.js` static file. Using launcher code allows you to update the bookmarklet code at any time without requiring users to change the bookmark they previously added to their browser.

Let's add the bookmarklet launcher to the dashboard pages so that users can add it to the bookmarks bar of their browser.

Edit the `account/dashboard.html` template of the `account` application and make it look like the following. New lines are highlighted in bold:

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

{% block title %}Dashboard{% endblock %}

{% block content %}
  <h1>Dashboard</h1>
  {% with total_images_created=request.user.images_created.count %}
    <p>Welcome to your dashboard. You have bookmarked {{ total_images_created
}} image{{ total_images_created|pluralize }}.</p>
  {% endwith %}
  <p>Drag the following button to your bookmarks toolbar to bookmark images
from other websites → <a href="javascript:{% include "bookmarklet_launcher.js"
%}" class="button">Bookmark it</a></p>
  <p>You can also <a href="{% url "edit" %}">edit your profile</a> or <a
href="{% url "password_change" %}">change your password</a>.</p>
{% endblock %}
```

Make sure that no template tag is split into multiple lines; Django doesn't support multiple-line tags.

The dashboard now displays the total number of images bookmarked by the user. We have added a `{% with %}` template tag to create a variable with the total number of images bookmarked by the current user. We have included a link with an `href` attribute that contains the bookmarklet launcher script. This JavaScript code is loaded from the `bookmarklet_launcher.js` template.

Open `https://127.0.0.1:8000/account/` in your browser. You should see the following page:

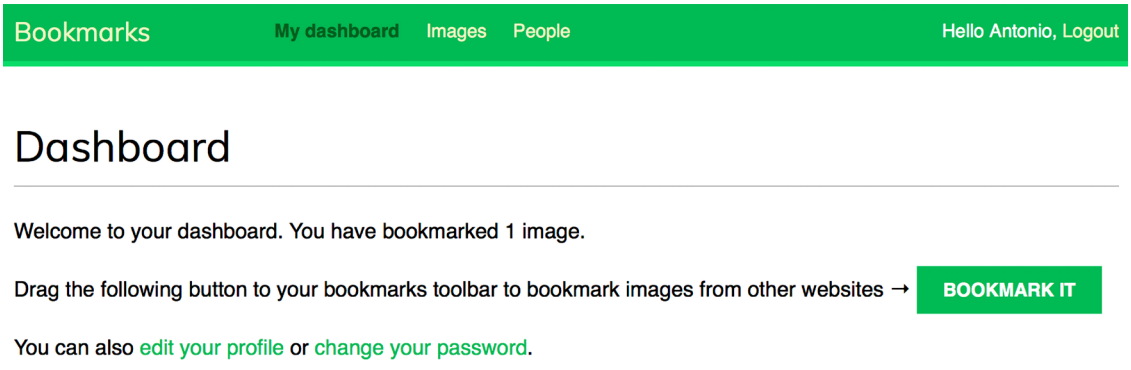


Figure 6.7: The dashboard page, including the total images bookmarked and the button for the bookmarklet

Now create the following directories and files inside the `images` application directory:

```
static/
  js/
    bookmarklet.js
```

You will find a `static/css/` directory under the `images` application directory in the code that comes along with this chapter. Copy the `css/` directory into the `static/` directory of your code. You can find the contents of the directory at `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter06/bookmarks/images/static`.

The `css/bookmarklet.css` file provides the styles for the JavaScript bookmarklet. The `static/` directory should contain the following file structure now:

```
  css/
    bookmarklet.css
  js/
    bookmarklet.js
```

Edit the `bookmarklet.js` static file and add the following JavaScript code to it:

```
const siteUrl = '//127.0.0.1:8000/';
const styleUrl = siteUrl + 'static/css/bookmarklet.css';
const minWidth = 250;
const minHeight = 250;
```

You have declared four different constants that will be used by the bookmarklet. These constants are:

- `siteUrl` and `staticUrl`: The base URL for the website and the base URL for static files.
- `minWidth` and `minHeight`: The minimum width and height in pixels for the images that the bookmarklet will collect from the site. The bookmarklet will identify images that have at least `250px` width and `250px` height.

Edit the `bookmarklet.js` static file and add the following code highlighted in bold:

```
const siteUrl = '//127.0.0.1:8000/';
const styleUrl = siteUrl + 'static/css/bookmarklet.css';
const minWidth = 250;
const minHeight = 250;

// load CSS
var head = document.getElementsByTagName('head')[0];
var link = document.createElement('link');
link.rel = 'stylesheet';
link.type = 'text/css';
link.href = styleUrl + '?r=' + Math.floor(Math.random()*9999999999999999);
head.appendChild(link);
```

This section loads the CSS stylesheet for the bookmarklet. We use JavaScript to manipulate the **Document Object Model** (**DOM**). The DOM represents an HTML document in memory and it is created by the browser when a web page is loaded. The DOM is constructed as a tree of objects that comprise the structure and content of the HTML document.

The previous code generates an object equivalent to the following JavaScript code and appends it to the `<head>` element of the HTML page:

```
<link rel="stylesheet" type="text/css" href= "//127.0.0.1:8000/static/css/
bookmarklet.css?r=1234567890123456">
```

Let's review how this is done:

1. The `<head>` element of the site is retrieved with `document.getElementsByTagName()`. This function retrieves all HTML elements of the page with the given tag. By using `[0]` we access the first instance found. We access the first element because all HTML documents should have a single `<head>` element.
2. A `<link>` element is created with `document.createElement('link')`.
3. The `rel` and `type` attributes of the `<link>` element are set. This is equivalent to the HTML `<link rel="stylesheet" type="text/css">`.
4. The `href` attribute of the `<link>` element is set with the URL of the `bookmarklet.css` stylesheet. A 16-digit random number is used as a URL parameter to prevent the browser from loading the file from the cache.

5. The new `<link>` element is added to the `<head>` element of the HTML page using `head.appendChild(link)`.

Now we will create the HTML element to display a container on the website where the bookmarklet is executed. The HTML container will be used to display all images found on the site and let users choose the image they want to share. It will use the CSS styles defined in the `bookmarklet.css` stylesheet.

Edit the `bookmarklet.js` static file and add the following code highlighted in bold:

```
const siteUrl = '//127.0.0.1:8000/';
const styleUrl = siteUrl + 'static/css/bookmarklet.css';
const minWidth = 250;
const minHeight = 250;

// load CSS
var head = document.getElementsByTagName('head')[0];
var link = document.createElement('link');
link.rel = 'stylesheet';
link.type = 'text/css';
link.href = styleUrl + '?r=' + Math.floor(Math.random()*9999999999999999);
head.appendChild(link);

// load HTML
var body = document.getElementsByTagName('body')[0];
boxHtml = '
  <div id="bookmarklet">
    <a href="#" id="close">&times;</a>
    <h1>Select an image to bookmark:</h1>
    <div class="images"></div>
  </div>';
body.innerHTML += boxHtml;
```

With this code the `<body>` element of the DOM is retrieved and new HTML is added to it by modifying its property `innerHTML`. A new `<div>` element is added to the body of the page. The `<div>` container consists of the following elements:

- A link to close the container defined with `<a href="#" id="close">&times;</a>`.
- A title defined with `<h1>Select an image to bookmark:</h1>`.
- An `<div>` element to list the images found on the site defined with `<div class="images"></div>`. This container is initially empty and will be filled with the images found on the site.

The HTML container, including the previously loaded CSS styles, will look like *Figure 6.8*:

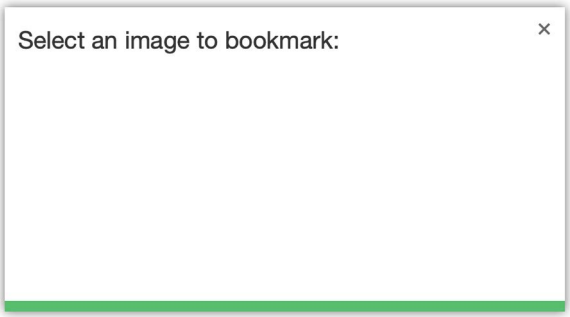


Figure 6.8: The image selection container

Now let's implement a function to launch the bookmarklet. Edit the `bookmarklet.js` static file and add the following code at the bottom:

```
function bookmarkletLaunch() {
  bookmarklet = document.getElementById('bookmarklet');
  var imagesFound = bookmarklet.querySelector('.images');

  // clear images found
  imagesFound.innerHTML = '';
  // display bookmarklet
  bookmarklet.style.display = 'block';

  // close event
  bookmarklet.querySelector('#close')
             .addEventListener('click', function(){
               bookmarklet.style.display = 'none'
             });
}

// launch the bookmkarklet
bookmarkletLaunch();
```

This is the `bookmarkletLaunch()` function. Before the definition of this function, the CSS for the bookmarklet is loaded and the HTML container is added to the DOM of the page. The `bookmarkletLaunch()` function works as follows:

1. The bookmarklet main container is retrieved by getting the DOM element with the ID `bookmarklet` with `document.getElementById()`.

2. The `bookmarklet` element is used to retrieve the child element with the class `images`. The `querySelector()` method allows you to retrieve DOM elements using CSS selectors. Selectors allow you to find DOM elements to which a set of CSS rules applies. You can find a list of CSS selectors at `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors` and you can read more information about how to locate DOM elements using selectors at `https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Locating_DOM_elements_using_selectors`.
3. The `images` container is cleared by setting its `innerHTML` attribute to an empty string and the bookmarklet is displayed by setting the `display` CSS property to `block`.
4. The `#close` selector is used to find the DOM element with the ID `close`. A `click` event is attached to the element with the `addEventListener()` method. When users click the element, the bookmarklet main container is hidden by setting its `display` property to `none`.

The `bookmarkletLaunch()` function is executed after its definition.

After loading the CSS styles and the HTML container of the bookmarklet, you have to find image elements in the DOM of the current website. Images that have the minimum required dimension have to be added to the HTML container of the bookmarklet. Edit the `bookmarklet.js` static file and add the following code highlighted in bold to the bottom of the `bookmarklet()` function:

```
function bookmarkletLaunch() {
  bookmarklet = document.getElementById('bookmarklet');
  var imagesFound = bookmarklet.querySelector('.images');

  // clear images found
  imagesFound.innerHTML = '';
  // display bookmarklet
  bookmarklet.style.display = 'block';

  // close event
  bookmarklet.querySelector('#close')
             .addEventListener('click', function(){
               bookmarklet.style.display = 'none'
             });

  // find images in the DOM with the minimum dimensions
  images = document.querySelectorAll('img[src$=".jpg"], img[src$=".jpeg"],
img[src$=".png"]');
  images.forEach(image => {
    if(image.naturalWidth >= minWidth
       && image.naturalHeight >= minHeight)
    {
      var imageFound = document.createElement('img');
      imageFound.src = image.src;
      imagesFound.append(imageFound);
```

```
    }
  })
}

// Launch the bookmarklet
bookmarkletLaunch();
```

The preceding code uses the `img[src$=".jpg"]`, `img[src$=".jpeg"]`, and `img[src$=".png"]` selectors to find all `<img>` DOM elements whose `src` attribute finishes with `.jpg`, `.jpeg`, or, `.png` respectively. Using these selectors with `document.querySelectorAll()` allows you to find all images with the JPEG and PNG format displayed on the website. Iteration over the results is performed with the `forEach()` method. Small images are filtered out because we don't consider them to be relevant. Only images with a size larger than the one specified with the `minWidth` and `minHeight` variables are used for the results. A new `<img>` element is created for each image found, where the `src` source URL attribute is copied from the original image and added to the `imagesFound` container.

For security reasons, your browser will prevent you from running the bookmarklet over HTTP on a site served through HTTPS. That's the reason we keep using RunServerPlus to run the development server using an auto-generated TLS/SSL certificate. Remember that you learned how to run the development server through HTTPS in *Chapter 5, Implementing Social Authentication*.

In a production environment, a valid TLS/SSL certificate will be required. When you own a domain name, you can apply for a trusted **Certification Authority** (CA) to issue a TLS/SSL certificate for it, so that browsers can verify its identity. If you want to obtain a trusted certificate for a real domain, you can use the *Let's Encrypt* service. *Let's Encrypt* is a nonprofit CA that simplifies obtaining and renewing trusted TLS/SSL certificates for free. You can find more information at `https://letsencrypt.org`.

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

```
python manage.py runserver_plus --cert-file cert.crt
```

Open `https://127.0.0.1:8000/account/` in your browser. Log in with an existing user, then click and drag the **BOOKMARK IT** button to the bookmarks bar of your browser, as follows:

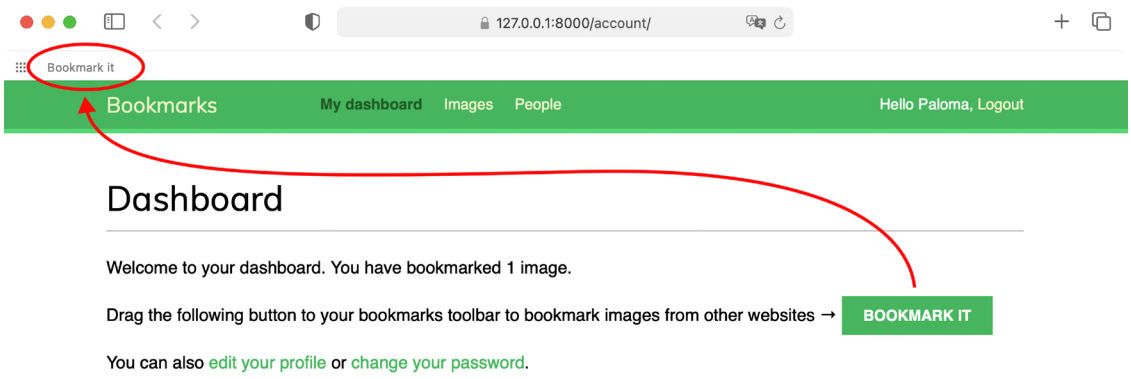


Figure 6.9: Adding the BOOKMARK IT button to the bookmarks bar

Open a website of your own choice in your browser and click on the **Bookmark it** bookmarklet in the bookmarks bar. You will see that a new white overlay appears on the website, displaying all JPEG and PNG images found with dimensions higher than 250×250 pixels. *Figure 6.10* shows the bookmarklet running on `https://amazon.com/`:

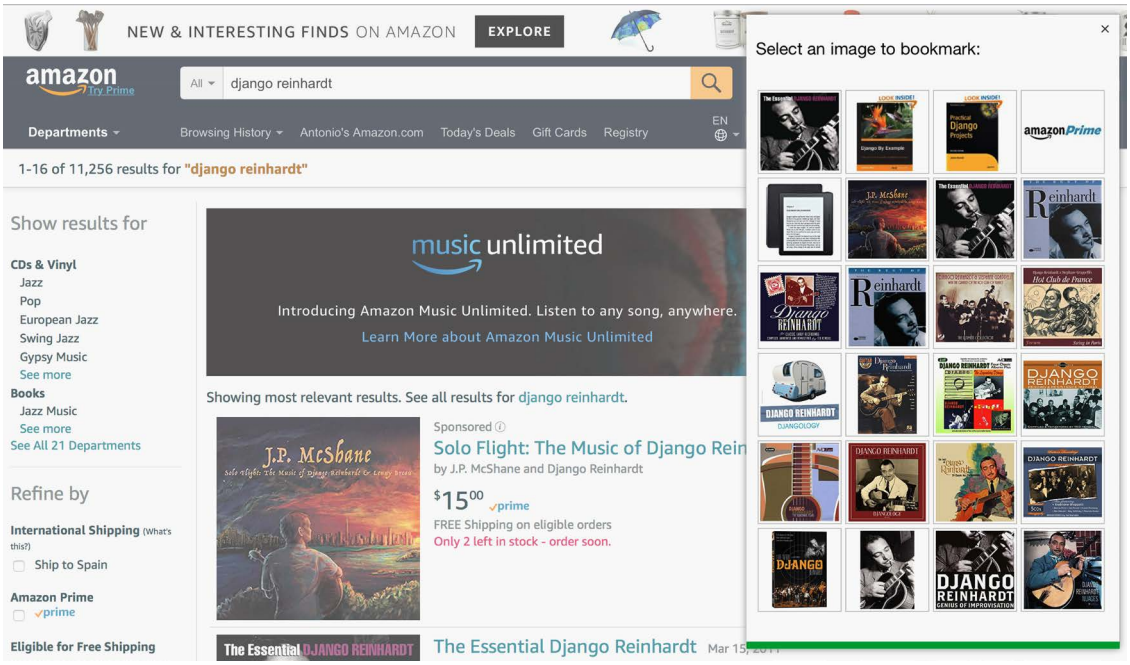


Figure 6.10: The bookmarklet loaded on amazon.com

If the HTML container doesn't appear, check the RunServer shell console log. If you see a MIME type error, it is most likely that your MIME map files are incorrect or need to be updated. You can apply the correct mapping for JavaScript and CSS files by adding the following lines to the `settings.py` file:

```
if DEBUG:
    import mimetypes
    mimetypes.add_type('application/javascript', '.js', True)
    mimetypes.add_type('text/css', '.css', True)
```

The HTML container includes the images that can be bookmarked. We will now implement the functionality for users to click on the desired image to bookmark it.

Edit the `js/bookmarklet.js` static file and add the following code at the bottom of the `bookmarklet()` function:

```
function bookmarkletLaunch() {
  bookmarklet = document.getElementById('bookmarklet');
  var imagesFound = bookmarklet.querySelector('.images');
```

```
  // clear images found
  imagesFound.innerHTML = '';
  // display bookmarklet
  bookmarklet.style.display = 'block';

  // close event
  bookmarklet.querySelector('#close')
             .addEventListener('click', function(){
               bookmarklet.style.display = 'none'
             });

  // find images in the DOM with the minimum dimensions
  images = document.querySelectorAll('img[src$=".jpg"], img[src$=".jpeg"],
img[src$=".png"]');
  images.forEach(image => {
    if(image.naturalWidth >= minWidth
       && image.naturalHeight >= minHeight)
    {
      var imageFound = document.createElement('img');
      imageFound.src = image.src;
      imagesFound.append(imageFound);
    }
  })

  // select image event
  imagesFound.querySelectorAll('img').forEach(image => {
    image.addEventListener('click', function(event){
      imageSelected = event.target;
      bookmarklet.style.display = 'none';
      window.open(siteUrl + 'images/create/?url='
                  + encodeURIComponent(imageSelected.src)
                  + '&title='
                  + encodeURIComponent(document.title),
                  '_blank');
    })
  })
}

// launch the bookmkarklet
bookmarkletLaunch();
```

The preceding code works as follows:

1. A `click()` event is attached to each image element within the `imagesFound` container.
2. When the user clicks on any of the images, the image element clicked is stored in the variable `imageSelected`.
3. The bookmarklet is then hidden by setting its `display` property to `none`.
4. A new browser window is opened with the URL to bookmark a new image on the site. The content of the `<title>` element of the website is passed to the URL in the `title` `GET` parameter and the selected image URL is passed in the `url` parameter.

Open a new URL with your browser, for example, `https://commons.wikimedia.org/`, as follows:

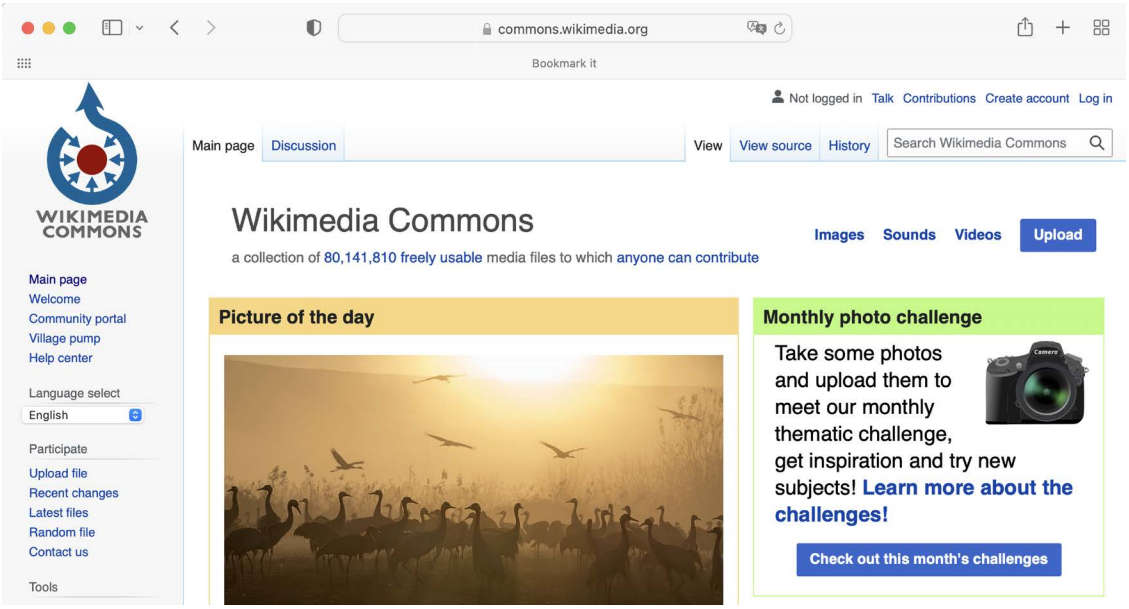


Figure 6.11: The Wikimedia Commons website

Figures 6.11 to *6.14* image: *A flock of cranes (Grus grus) in Hula Valley, Northern Israel* by Tomere (Licence: Creative Commons Attribution-Share Alike 4.0 International: `https://creativecommons.org/licenses/by-sa/4.0/deed.en`)

Click on the **Bookmark it** bookmarklet to display the image selection overlay. You will see the image selection overlay like this:

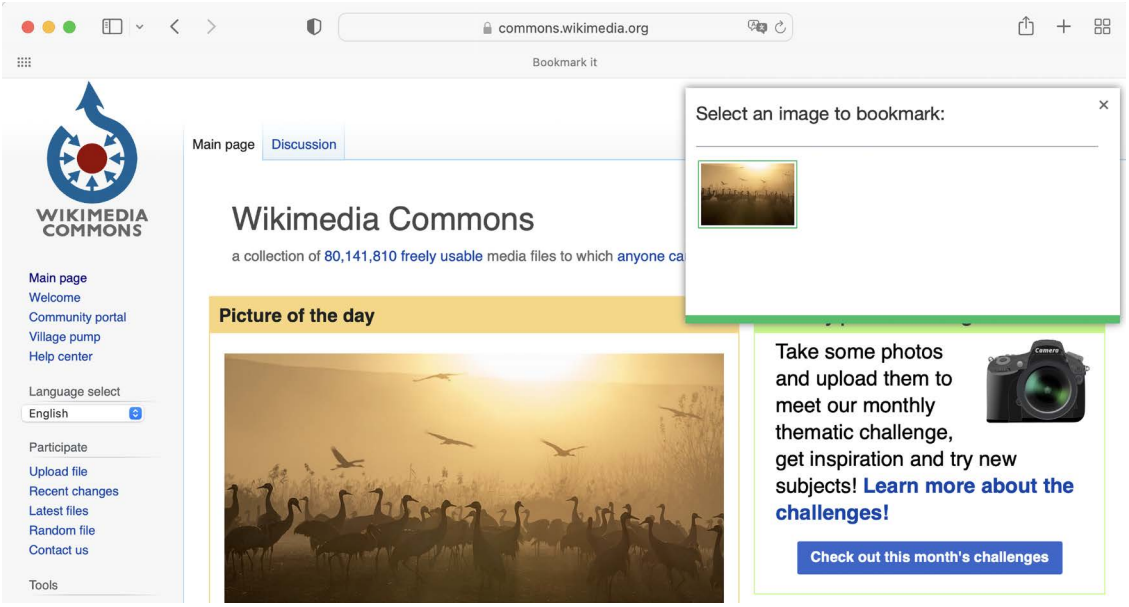


Figure 6.12: The bookmarklet loaded on an external website

If you click on an image, you will be redirected to the image creation page, passing the title of the website and the URL of the selected image as `GET` parameters. The page will look as follows:

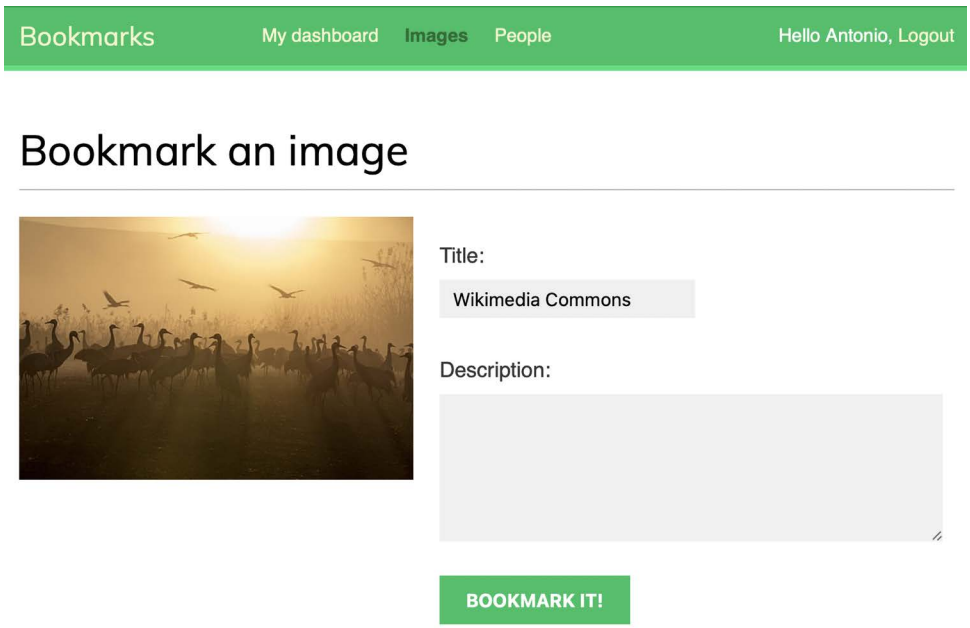


Figure 6.13: The form to bookmark an image

Congratulations! This is your first JavaScript bookmarklet, and it is fully integrated into your Django project. Next, we will create the detail view for images and implement the canonical URL for images.

Creating a detail view for images

Let's now create a simple detail view to display images that have been bookmarked on the site. Open the `views.py` file of the `images` application and add the following code to it:

```
from django.shortcuts import get_object_or_404
from .models import Image

def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image})
```

This is a simple view to display an image. Edit the `urls.py` file of the `images` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>/',
         views.image_detail, name='detail'),
]
```

Edit the `models.py` file of the `images` application and add the `get_absolute_url()` method to the `Image` model, as follows:

```
from django.urls import reverse

class Image(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('images:detail', args=[self.id,
                                              self.slug])
```

Remember that the common pattern for providing canonical URLs for objects is to define a `get_absolute_url()` method in the model.

Finally, create a template inside the `/templates/images/image/` template directory for the `images` application and name it `detail.html`. Add the following code to it:

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

{% block title %}{{ image.title }}{% endblock %}

{% block content %}
  <h1>{{ image.title }}</h1>
  <img src="{{ image.image.url }}" class="image-detail">
  {% with total_likes=image.users_like.count %}
    <div class="image-info">
      <div>
        <span class="count">
          {{ total_likes }} like{{ total_likes|pluralize }}
        </span>
      </div>
      {{ image.description|linebreaks }}
    </div>
    <div class="image-likes">
      {% for user in image.users_like.all %}
        <div>
          {% if user.profile.photo %}
            <img src="{{ user.profile.photo.url }}">
          {% endif %}
          <p>{{ user.first_name }}</p>
        </div>
      {% empty %}
        Nobody likes this image yet.
      {% endfor %}
    </div>
  {% endwith %}
{% endblock %}
```

This is the template to display the detail view of a bookmarked image. We have used the `{% with %}` tag to create the `total_likes` variable with the result of a QuerySet that counts all user likes. By doing so, we avoid evaluating the same QuerySet twice (first to display the total number of likes, then to use the `pluralize` template filter). We have also included the image description and we have added a `{% for %}` loop to iterate over `image.users_like.all` to display all the users who like this image.

Whenever you need to repeat a query in your template, use the `{% with %}` template tag to avoid additional database queries.

Now, open an external URL in your browser and use the bookmarklet to bookmark a new image. You will be redirected to the image detail page after you post the image. The page will include a success message, as follows:

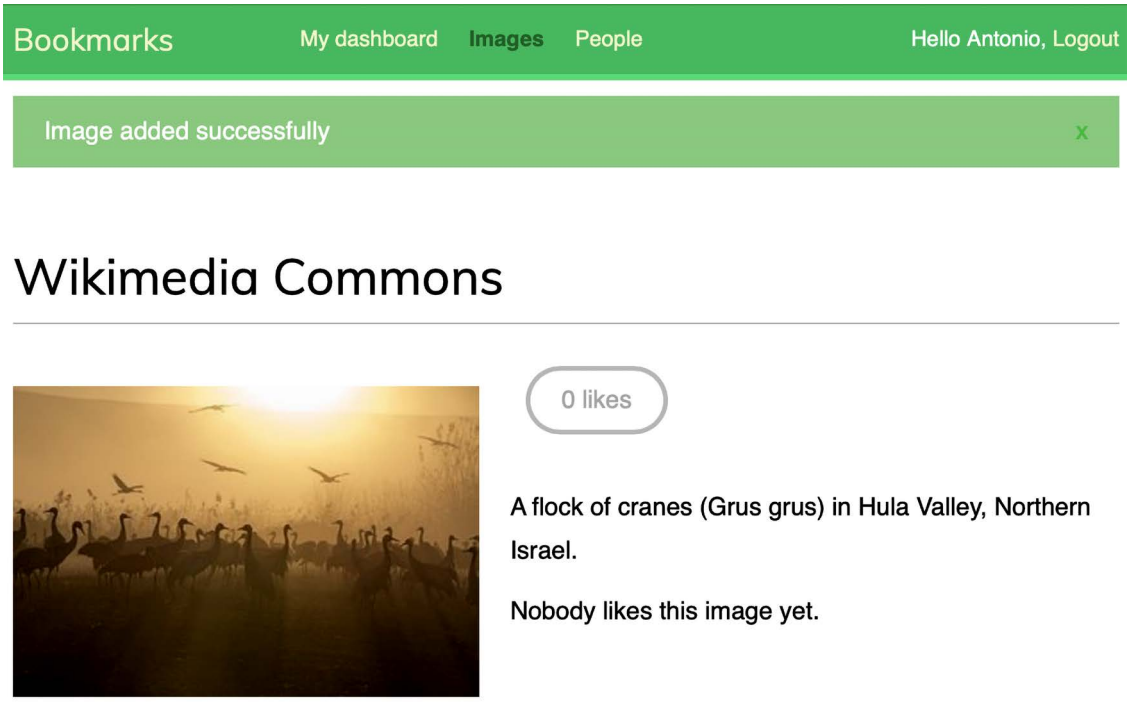


Figure 6.14: The image detail page for the image bookmark

Great! You completed the bookmarklet functionality. Next, you will learn how to create thumbnails for images.

Creating image thumbnails using easy-thumbnails

We are displaying the original image on the detail page, but dimensions for different images may vary considerably. The file size for some images may be very large, and loading them might take too long. The best way to display optimized images in a uniform manner is to generate thumbnails. A thumbnail is a small image representation of a larger image. Thumbnails will load faster in the browser and are a great way to homogenize images of very different sizes. We will use a Django application called `easy-thumbnails` to generate thumbnails for the images bookmarked by users.

Open the terminal and install `easy-thumbnails` using the following command:

```
pip install easy-thumbnails==2.8.1
```

Edit the `settings.py` file of the `bookmarks` project and add `easy_thumbnails` to the `INSTALLED_APPS` setting, as follows:

```
INSTALLED_APPS = [
    # ...
    'easy_thumbnails',
]
```

Then, run the following command to sync the application with your database:

```
python manage.py migrate
```

You will see an output that includes the following lines:

```
Applying easy_thumbnails.0001_initial... OK
Applying easy_thumbnails.0002_thumbnaildimensions... OK
```

The `easy-thumbnails` application offers you different ways to define image thumbnails. The application provides a `{% thumbnail %}` template tag to generate thumbnails in templates and a custom `ImageField` if you want to define thumbnails in your models. Let's use the template tag approach.

Edit the `images/image/detail.html` template and consider the following line:

```
<img src="{{ image.image.url }}" class="image-detail">
```

The following lines should replace the preceding one:

```
{% load thumbnail %}
<a href="{{ image.image.url }}">
  <img src="{% thumbnail image.image 300x0 %}" class="image-detail">
</a>
```

We have defined a thumbnail with a fixed width of `300` pixels and a flexible height to maintain the aspect ratio by using the value `0`. The first time a user loads this page, a thumbnail image will be created. The thumbnail is stored in the same directory as the original file. The location is defined by the `MEDIA_ROOT` setting and the `upload_to` attribute of the `image` field of the `Image` model. The generated thumbnail will then be served in the following requests.

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

```
python manage.py runserver_plus --cert-file cert.crt
```

Access the image detail page for an existing image. The thumbnail will be generated and displayed on the site. Right-click on the image and open it in a new browser tab as follows:

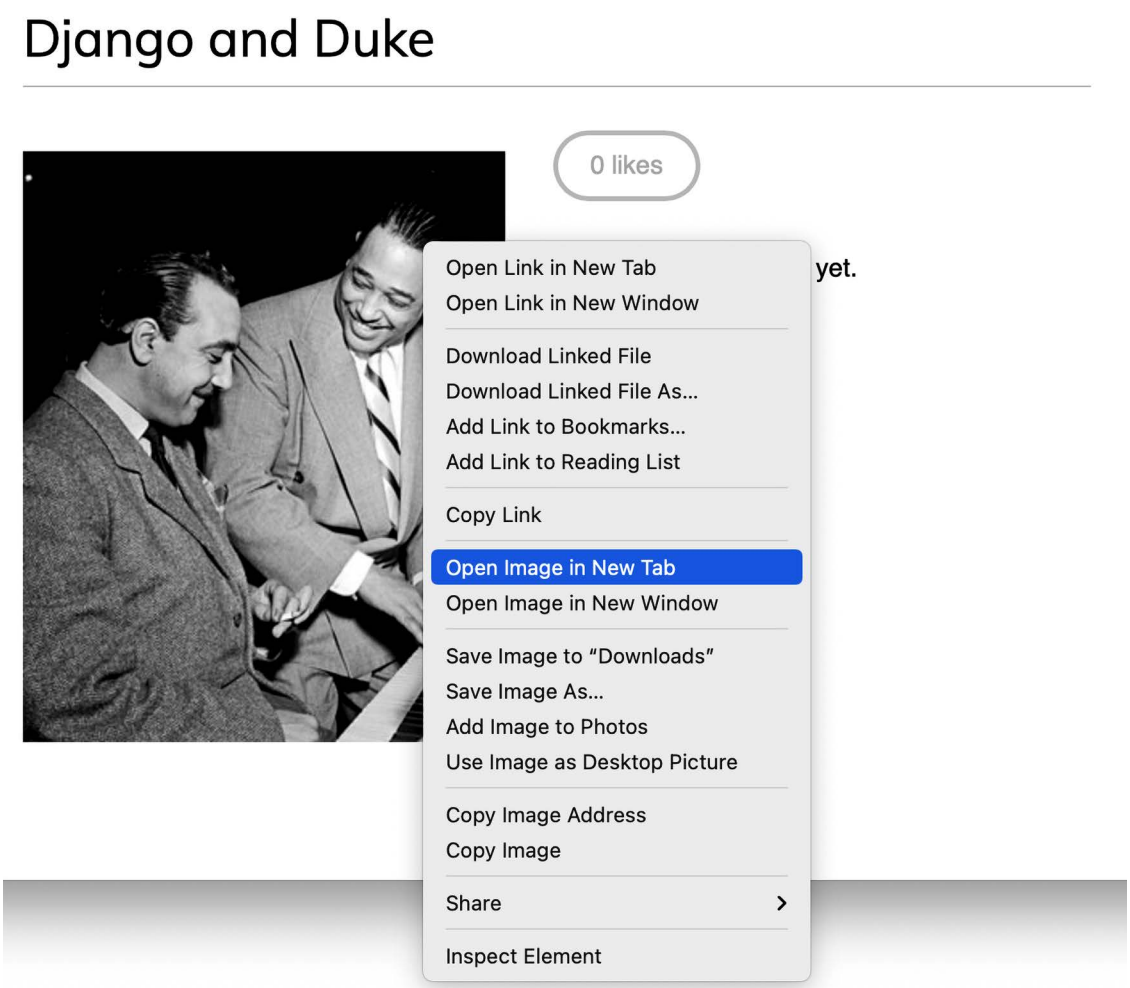


Figure 6.15: Open the image in a new browser tab

Check the URL of the generated image in your browser. It should look as follows:

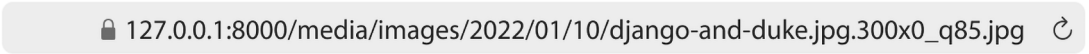


Figure 6.16: The URL of the generated image

The original filename is followed by additional details of the settings used to create the thumbnail. For a JPEG image, you will see a filename like `filename.jpg.300x0_q85.jpg`, where `300x0` are the size parameters used to generate the thumbnail, and `85` is the value for the default JPEG quality used by the library to generate the thumbnail.

You can use a different quality value using the `quality` parameter. To set the highest JPEG quality, you can use the value `100`, like this: `{% thumbnail image.image 300x0 quality=100 %}`. A higher quality will imply a larger file size.

The `easy-thumbnails` application offers several options to customize your thumbnails, including cropping algorithms and different effects that can be applied. If you run into any issues generating thumbnails, you can add `THUMBNAIL_DEBUG = True` to the `settings.py` file to obtain the debug information. You can read the full documentation of `easy-thumbnails` at `https://easy-thumbnails.readthedocs.io/`.

Adding asynchronous actions with JavaScript

We are going to add a *like* button to the image detail page to let users click on it to like an image. When users click the *like* button, we will send an HTTP request to the web server using JavaScript. This will perform the like action without reloading the whole page. For this functionality, we will implement a view that allows users to like/unlike images.

The JavaScript **Fetch API** is the built-in way to make asynchronous HTTP requests to web servers from web browsers. By using the Fetch API, you can send and retrieve data from the web server without the need for a whole page refresh. The Fetch API was launched as a modern successor to the browser built-in `XMLHttpRequest` (XHR) object, used to make HTTP requests without reloading the page. The set of web development techniques to send and retrieve data from a web server asynchronously without reloading the page is also known as **AJAX**, which stands for **Asynchronous JavaScript and XML**. AJAX is a misleading name because AJAX requests can exchange data not only in XML format but also in formats such as JSON, HTML, and plain text. You might find references to the Fetch API and AJAX indistinctively on the Internet.

You can find information about the Fetch API at `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch`.

We will start by implementing the view to perform the *like* and *unlike* actions, and then we will add the JavaScript code to the related template to perform asynchronous HTTP requests.

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

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

@login_required
@require_POST
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
```

```
            if action == 'like':
                image.users_like.add(request.user)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status': 'ok'})
        except Image.DoesNotExist:
            pass
    return JsonResponse({'status': 'error'})
```

We have used two decorators for the new view. The `login_required` decorator prevents users who are not logged in from accessing this view. The `require_POST` decorator returns an `HttpResponseNotAllowed` object (status code `405`) if the HTTP request is not done via `POST`. This way, you only allow `POST` requests for this view.

Django also provides a `require_GET` decorator to only allow `GET` requests and a `require_http_methods` decorator to which you can pass a list of allowed methods as an argument.

This view expects the following `POST` parameters:

- `image_id`: The ID of the `image` object on which the user is performing the action
- `action`: The action that the user wants to perform, which should be a string with the value `like` or `unlike`

We have used the manager provided by Django for the `users_like` many-to-many field of the `Image` model in order to add or remove objects from the relationship using the `add()` or `remove()` methods. If the `add()` method is called passing an object that is already present in the related object set, it will not be duplicated. If the `remove()` method is called with an object that is not in the related object set, nothing will happen. Another useful method of many-to-many managers is `clear()`, which removes all objects from the related object set.

To generate the view response, we have used the `JsonResponse` class provided by Django, which returns an HTTP response with an `application/json` content type, converting the given object into a JSON output.

Edit the `urls.py` file of the `images` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>/',
         views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
]
```

Loading JavaScript on the DOM

We need to add JavaScript code to the image detail template. To use JavaScript in our templates, we will add a base wrapper in the `base.html` template of the project first.

Edit the `base.html` template of the `account` application and include the following code highlighted in bold before the closing `</body>` HTML tag:

```
<!DOCTYPE html>
<html>
<head>
 ...
</head>
<body>
  ...
  <script>
    document.addEventListener('DOMContentLoaded', (event) => {
      // DOM loaded
      {% block domready %}
      {% endblock %}
    })
  </script>
</body>
</html>
```

We have added a `<script>` tag to include JavaScript code. The `document.addEventListener()` method is used to define a function that will be called when the given event is triggered. We pass the event name `DOMContentLoaded`, which fires when the initial HTML document has been completely loaded and the **Document Object Model** (**DOM**) hierarchy has been fully constructed. By using this event, we make sure the DOM is fully constructed before we interact with any HTML elements and we manipulate the DOM. The code within the function will only be executed once the DOM is ready.

Inside the document-ready handler, we have included a Django template block called `domready`. Any template that extends the `base.html` template can use this block to include specific JavaScript code to execute when the DOM is ready.

Don't get confused by the JavaScript code and Django template tags. The Django template language is rendered on the server side to generate the HTML document, and JavaScript is executed in the browser on the client side. In some cases, it is useful to generate JavaScript code dynamically using Django, to be able to use the results of QuerySets or server-side calculations to define variables in JavaScript.

The examples in this chapter include JavaScript code in Django templates. The preferred method to add JavaScript code to your templates is by loading `.js` files, which are served as static files, especially if you are using large scripts.

Cross-site request forgery for HTTP requests in JavaScript

You learned about **cross-site request forgery** (**CSRF**) in *Chapter 2*, *Enhancing Your Blog with Advanced Features*. With CSRF protection active, Django looks for a CSRF token in all `POST` requests. When you submit forms, you can use the `{% csrf_token %}` template tag to send the token along with the form. HTTP requests made in JavaScript have to pass the CSRF token as well in every `POST` request.

Django allows you to set a custom `X-CSRFToken` header in your HTTP requests with the value of the CSRF token.

To include the token in HTTP requests that originate from JavaScript, we will need to retrieve the CSRF token from the `csrftoken` cookie, which is set by Django if the CSRF protection is active. To handle cookies, we will use the JavaScript Cookie library. JavaScript Cookie is a lightweight JavaScript API for handling cookies. You can learn more about it at `https://github.com/js-cookie/js-cookie`.

Edit the `base.html` template of the `account` application and add the following code highlighted in bold at the bottom of the `<body>` element like this:

```
<!DOCTYPE html>
<html>
<head>
 ...
</head>
<body>
  ...
  <script src="//cdn.jsdelivr.net/npm/js-cookie@3.0.1/dist/js.cookie.min.js"></
script>
  <script>
    const csrftoken = Cookies.get('csrftoken');
    document.addEventListener('DOMContentLoaded', (event) => {
      // DOM loaded
      {% block domready %}
      {% endblock %}
    })
  </script>
</body>
</html>
```

We have implemented the following functionality:

1. The JS Cookie plugin is loaded from a public **Content Delivery Network** (**CDN**).
2. The value of the `csrftoken` cookie is retrieved with `Cookies.get()` and stored in the JavaScript constant `csrftoken`.

We have to include the CSRF token in all JavaScript fetch requests that use unsafe HTTP methods, such as `POST` or `PUT`. We will later include the `csrftoken` constant in a custom HTTP header named `X-CSRFToken` when sending HTTP `POST` requests.

You can find more information about Django's CSRF protection and AJAX at `https://docs.djangoproject.com/en/4.1/ref/csrf/#ajax`.

Next, we will implement the HTML and JavaScript code for users to like/unlike images.

Performing HTTP requests with JavaScript

Edit the `images/image/detail.html` template and add the following code highlighted in bold:

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

{% block title %}{{ image.title }}{% endblock %}

{% block content %}
  <h1>{{ image.title }}</h1>
  {% load thumbnail %}
  <a href="{{ image.image.url }}">
    <img src="{% thumbnail image.image 300x0 %}" class="image-detail">
  </a>
  {% with total_likes=image.users_like.count users_like=image.users_like.all %}
    <div class="image-info">
      <div>
        <span class="count">
        <span class="total">{{ total_likes }}</span>
        like{{ total_likes|pluralize }}
        </span>
        <a href="#" data-id="{{ image.id }}" data-action="{% if request.user in
users_like %}un{% endif %}like"
    class="like button">
          {% if request.user not in users_like %}
            Like
          {% else %}
            Unlike
          {% endif %}
        </a>
      </div>
      {{ image.description|linebreaks }}
    </div>
    <div class="image-likes">
      {% for user in users_like %}
        <div>
          {% if user.profile.photo %}
            <img src="{{ user.profile.photo.url }}">
          {% endif %}
          <p>{{ user.first_name }}</p>
        </div>
      {% empty %}
```

```
        Nobody likes this image yet.
      {% endfor %}
    </div>
  {% endwith %}
{% endblock %}
```

In the preceding code, we have added another variable to the `{% with %}` template tag to store the results of the `image.users_like.all` query and avoid executing the query against the database multiple times. This variable is used to check if the current user is in this list with `{% if request.user in users_like %}` and then with `{% if request.user not in users_like %}`. The same variable is then used to iterate over the users that like this image with `{% for user in users_like %}`.

We have added to this page the total number of users who like the image and have included a link for the user to like/unlike the image. The related object set, `users_like`, is used to check whether `request.user` is contained in the related object set, to display the text *Like* or *Unlike* based on the current relationship between the user and this image. We have added the following attributes to the `<a>` HTML link element:

- `data-id`: The ID of the image displayed.
- `data-action`: The action to perform when the user clicks on the link. This can be either `like` or `unlike`.

Any attribute on any HTML element with a name that starts with `data-` is a data attribute. Data attributes are used to store custom data for your application.

We will send the value of the `data-id` and `data-action` attributes in the HTTP request to the `image_like` view. When a user clicks on the `like/unlike` link, we will need to perform the following actions in the browser:

1. Send an HTTP `POST` request to the `image_like` view, passing the image `id` and the `action` parameters to it.
2. If the HTTP request is successful, update the `data-action` attribute of the `<a>` HTML element with the opposite action (`like` / `unlike`), and modify its display text accordingly.
3. Update the total number of `likes` displayed on the page.

Add the following `domready` block at the bottom of the `images/image/detail.html` template:

```
{% block domready %}
  const url = '{% url "images:like" %}';
  var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrftoken},
    mode: 'same-origin'
```

```
  }

  document.querySelector('a.like')
          .addEventListener('click', function(e){
    e.preventDefault();
    var likeButton = this;
  });
{% endblock %}
```

The preceding code works as follows:

1. The `{% url %}` template tag is used to build the `images:like` URL. The generated URL is stored in the `url` JavaScript constant.
2. An `options` object is created with the options that will be passed to the HTTP request with the Fetch API. These are:
    - `method`: The HTTP method to use. In this case, it's `POST`.
    - `headers`: Additional HTTP headers to include in the request. We include the `X-CSRFToken` header with the value of the `csrftoken` constant that we defined in the `base.html` template.
    - `mode`: The mode of the HTTP request. We use `same-origin` to indicate the request is made to the same origin. You can find more information about modes at `https://developer.mozilla.org/en-US/docs/Web/API/Request/mode`.
3. The `a.like` selector is used to find all `<a>` elements of the HTML document with the `like` class using `document.querySelector()`.
4. An event listener is defined for the `click` event on the elements targeted with the selector. This function is executed every time the user clicks on the `like/unlike` link.
5. Inside the handler function, `e.preventDefault()` is used to avoid the default behavior of the `<a>` element. This will prevent the default behavior of the link element, stopping the event propagation, and preventing the link from following the URL.
6. A variable `likeButton` is used to store the reference to `this`, the element on which the event was triggered.

Now we need to send the HTTP request using the Fetch API. Edit the `domready` block of the `images/image/detail.html` template and add the following code highlighted in bold:

```
{% block domready %}
  const url = '{% url "images:like" %}';
  var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrftoken},
    mode: 'same-origin'
  }
```

```
  document.querySelector('a.like')
          .addEventListener('click', function(e){
    e.preventDefault();
    var likeButton = this;

    // add request body
    var formData = new FormData();
    formData.append('id', likeButton.dataset.id);
    formData.append('action', likeButton.dataset.action);
    options['body'] = formData;

    // send HTTP request
    fetch(url, options)
    .then(response => response.json())
    .then(data => {
      if (data['status'] === 'ok')
      {
      }
    })
  });
{% endblock %}
```

The new code works as follows:

1. A `FormData` object is created to construct a set of key/value pairs representing form fields and their values. The object is stored in the `formData` variable.
2. The `id` and `action` parameters expected by the `image_like` Django view are added to the `formData` object. The values for these parameters are retrieved from the `likeButton` element clicked. The `data-id` and `data-action` attributes are accessed with `dataset.id` and `dataset.action`.
3. A new `body` key is added to the `options` object that will be used for the HTTP request. The value for this key is the `formData` object.
4. The Fetch API is used by calling the `fetch()` function. The `url` variable defined previously is passed as the URL for the request, and the `options` object is passed as the options for the request.
5. The `fetch()` function returns a promise that resolves with a `Response` object, which is a representation of the HTTP response. The `.then()` method is used to define a handler for the promise. To extract the JSON body content we use `response.json()`. You can learn more about the `Response` object at `https://developer.mozilla.org/en-US/docs/Web/API/Response`.
6. The `.then()` method is used again to define a handler for the data extracted to JSON. In this handler, the `status` attribute of the data received is used to check whether its value is `ok`.

You added the functionality to send the HTTP request and handle the response. After a successful request, you need to change the button and its related action to the opposite: from *like* to *unlike*, or from *unlike* to *like*. By doing so, users are able to undo their action.

Edit the `domready` block of the `images/image/detail.html` template and add the following code highlighted in bold:

```
{% block domready %}
  var url = '{% url "images:like" %}';
  var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrftoken},
    mode: 'same-origin'
  }

  document.querySelector('a.like')
          .addEventListener('click', function(e){
    e.preventDefault();
    var likeButton = this;

    // add request body
    var formData = new FormData();
    formData.append('id', likeButton.dataset.id);
    formData.append('action', likeButton.dataset.action);
    options['body'] = formData;

    // send HTTP request
    fetch(url, options)
    .then(response => response.json())
    .then(data => {
      if (data['status'] === 'ok')
      {
        var previousAction = likeButton.dataset.action;

        // toggle button text and data-action
        var action = previousAction === 'like' ? 'unlike' : 'like';
        likeButton.dataset.action = action;
        likeButton.innerHTML = action;

        // update like count
        var likeCount = document.querySelector('span.count .total');
        var totalLikes = parseInt(likeCount.innerHTML);
```

```
        likeCount.innerHTML = previousAction === 'like' ? totalLikes + 1 :
totalLikes - 1;
      }
    })
  });
{% endblock %}
```

The preceding code works as follows:

1. The previous action of the button is retrieved from the `data-action` attribute of the link and it is stored in the `previousAction` variable.
2. The `data-action` attribute of the link and the link text are toggled. This allows users to undo their action.
3. The total like count is retrieved from the DOM by using the selector `span.count.total` and the value is parsed to an integer with `parseInt()`. The total like count is increased or decreased according to the action performed (*like* or *unlike*).

Open the image detail page in your browser for an image that you have uploaded. You should be able to see the following initial likes count and the **LIKE** button, as follows:

Figure 6.17: The likes count and LIKE button in the image detail template

Click on the **LIKE** button. You will note that the total likes count increases by one and the button text changes to **UNLIKE**, as follows:

Figure 6.18: The likes count and button after clicking the LIKE button

If you click on the **UNLIKE** button, the action is performed, and then the button's text changes back to **LIKE** and the total count changes accordingly.

When programming JavaScript, especially when performing AJAX requests, it is recommended to use a tool for debugging JavaScript and HTTP requests. Most modern browsers include developer tools to debug JavaScript. Usually, you can right-click anywhere on the website to open the contextual menu and click on **Inspect** or **Inspect Element** to access the web developer tools of your browser.

In the next section, you will learn how to use asynchronous HTTP requests with JavaScript and Django to implement infinite scroll pagination.

Adding infinite scroll pagination to the image list

Next, we need to list all bookmarked images on the website. We will use JavaScript requests to build an infinite scroll functionality. Infinite scroll is achieved by loading the next results automatically when the user scrolls to the bottom of the page.

Let's implement an image list view that will handle both standard browser requests and requests originating from JavaScript. When the user initially loads the image list page, we will display the first page of images. When they scroll to the bottom of the page, we will retrieve the following page of items with JavaScript and append it to the bottom of the main page.

The same view will handle both standard and AJAX infinite scroll pagination. Edit the `views.py` file of the `images` application and add the following code highlighted in bold:

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

# ...

@login_required
def image_list(request):
    images = Image.objects.all()
    paginator = Paginator(images, 8)
    page = request.GET.get('page')
    images_only = request.GET.get('images_only')
    try:
        images = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer deliver the first page
        images = paginator.page(1)
    except EmptyPage:
        if images_only:
            # If AJAX request and page out of range
            # return an empty page
            return HttpResponse('')
        # If page out of range return last page of results
        images = paginator.page(paginator.num_pages)
    if images_only:
        return render(request,
                      'images/image/list_images.html',
                      {'section': 'images',
                       'images': images})
```

```
    return render(request,
                  'images/image/list.html',
                   {'section': 'images',
                    'images': images})
```

In this view, a QuerySet is created to retrieve all images from the database. Then, a `Paginator` object is created to paginate over the results, retrieving eight images per page. The `page` HTTP `GET` parameter is retrieved to get the requested page number. The `images_only` HTTP `GET` parameter is retrieved to know if the whole page has to be rendered or only the new images. We will render the whole page when it is requested by the browser. However, we will only render the HTML with new images for Fetch API requests, since we will be appending them to the existing HTML page.

An `EmptyPage` exception will be triggered if the requested page is out of range. If this is the case and only images have to be rendered, an empty `HttpResponse` will be returned. This will allow you to stop the AJAX pagination on the client side when reaching the last page. The results are rendered using two different templates:

- For JavaScript HTTP requests, that will include the `images_only` parameter, the `list_images.html` template will be rendered. This template will only contain the images of the requested page.
- For browser requests, the `list.html` template will be rendered. This template will extend the `base.html` template to display the whole page and will include the `list_images.html` template to include the list of images.

Edit the `urls.py` file of the `images` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>/',
         views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
    path('', views.image_list, name='list'),
]
```

Finally, you need to create the templates mentioned here. Inside the `images/image/` template directory, create a new template and name it `list_images.html`. Add the following code to it:

```
{% load thumbnail %}
{% for image in images %}
  <div class="image">
    <a href="{{ image.get_absolute_url }}">
      {% thumbnail image.image 300x300 crop="smart" as im %}
      <a href="{{ image.get_absolute_url }}">
        <img src="{{ im.url }}">
      </a>
    </a>
```

```
    <div class="info">
      <a href="{{ image.get_absolute_url }}" class="title">
        {{ image.title }}
      </a>
    </div>
  </div>
{% endfor %}
```

The preceding template displays the list of images. You will use it to return results for AJAX requests. In this code, you iterate over images and generate a square thumbnail for each image. You normalize the size of the thumbnails to `300x300` pixels. You also use the `smart` cropping option. This option indicates that the image has to be incrementally cropped down to the requested size by removing slices from the edges with the least entropy.

Create another template in the same directory and name it `images/image/list.html`. Add the following code to it:

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

{% block title %}Images bookmarked{% endblock %}

{% block content %}
  <h1>Images bookmarked</h1>
  <div id="image-list">
    {% include "images/image/list_images.html" %}
  </div>
{% endblock %}
```

The list template extends the `base.html` template. To avoid repeating code, you include the `images/image/list_images.html` template for displaying images. The `images/image/list.html` template will hold the JavaScript code for loading additional pages when scrolling to the bottom of the page.

Edit the `images/image/list.html` template and add the following code highlighted in bold:

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

{% block title %}Images bookmarked{% endblock %}

{% block content %}
  <h1>Images bookmarked</h1>
  <div id="image-list">
    {% include "images/image/list_images.html" %}
  </div>
{% endblock %}
```

```
{% block domready %}
  var page = 1;
  var emptyPage = false;
  var blockRequest = false;

  window.addEventListener('scroll', function(e) {
    var margin = document.body.clientHeight - window.innerHeight - 200;
    if(window.pageYOffset > margin && !emptyPage && !blockRequest) {
      blockRequest = true;
      page += 1;

      fetch('?images_only=1&page=' + page)
      .then(response => response.text())
      .then(html => {
        if (html === '') {
          emptyPage = true;
        }
        else {
          var imageList = document.getElementById('image-list');
          imageList.insertAdjacentHTML('beforeEnd', html);
          blockRequest = false;
        }
      })
    }
  });

  // Launch scroll event
  const scrollEvent = new Event('scroll');
  window.dispatchEvent(scrollEvent);
{% endblock %}
```

The preceding code provides the infinite scroll functionality. You include the JavaScript code in the `domready` block that you defined in the `base.html` template. The code is as follows:

1. You define the following variables:
    - `page`: Stores the current page number.
    - `empty_page`: Allows you to know whether the user is on the last page and retrieves an empty page. As soon as you get an empty page, you will stop sending additional HTTP requests because you will assume that there are no more results.
    - `block_request`: Prevents you from sending additional requests while an HTTP request is in progress.

2. You use `window.addEventListener()` to capture the `scroll` event and to define a handler function for it.
3. You calculate the `margin` variable to get the difference between the total document height and the window inner height, because that's the height of the remaining content for the user to scroll. You subtract a value of `200` from the result so that you load the next page when the user is closer than 200 pixels to the bottom of the page.
4. Before sending an HTTP request, you check that:
    - The offset `window.pageYOffset` is higher than the calculated margin.
    - The user didn't get to the last page of results (`emptyPage` has to be `false`).
    - There is no other ongoing HTTP request (`blockRequest` has to be `false`).
5. If the previous conditions are met, you set `blockRequest` to `true` to prevent the `scroll` event from triggering additional HTTP requests, and you increase the `page` counter by `1` to retrieve the next page.
6. You use `fetch()` to send an HTTP `GET` request, setting the URL parameters `image_only=1` to retrieve only the HTML for images instead of the whole HTML page, and `page` for the requested page number.
7. The body content is extracted from the HTTP response with `response.text()` and the HTML returned is treated accordingly:
    - **If the response has no content**: You got to the end of the results, and there are no more pages to load. You set `emptyPage` to `true` to prevent additional HTTP requests.
    - **If the response contains data**: You append the data to the HTML element with the `image-list` ID. The page content expands vertically, appending results when the user approaches the bottom of the page. You remove the lock for additional HTTP requests by setting `blockRequest` to `false`.
8. Below the event listener, you simulate an initial `scroll` event when the page is loaded. You create the event by creating a new `Event` object, and then you launch it with `window.dispatchEvent()`. By doing this, you ensure that the event is triggered if the initial content fits the window and has no scroll.

Open `https://127.0.0.1:8000/images/` in your browser. You will see the list of images that you have bookmarked so far. It should look similar to this:

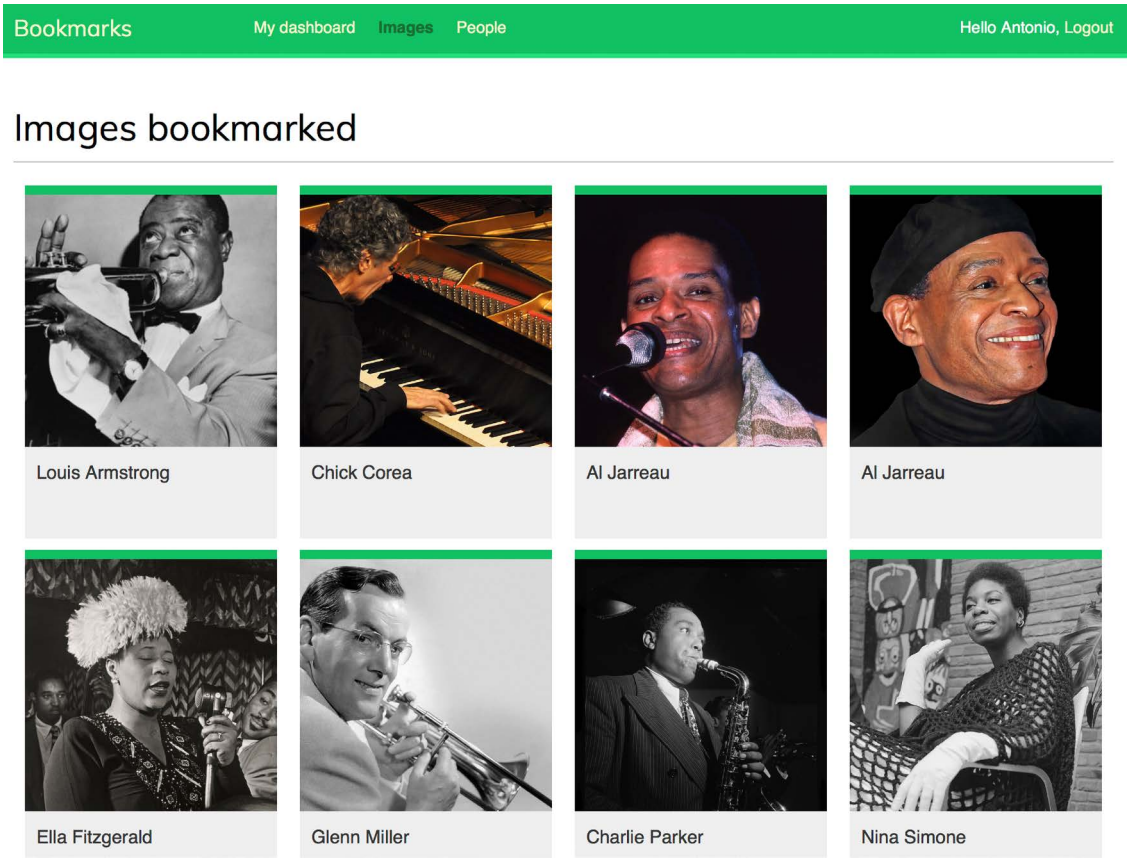


Figure 6.19: The image list page with infinite scroll pagination

Figure 6.19 image attributions:

- *Chick Corea* by ataelw (license: Creative Commons Attribution 2.0 Generic: `https://creativecommons.org/licenses/by/2.0/`)
- *Al Jarreau – Düsseldorf 1981* by Eddi Laumanns aka RX-Guru (license: Creative Commons Attribution 3.0 Unported: `https://creativecommons.org/licenses/by/3.0/`)
- *Al Jarreau* by Kingkongphoto & www.celebrity-photos.com (license: Creative Commons Attribution-ShareAlike 2.0 Generic: `https://creativecommons.org/licenses/by-sa/2.0/`)

Scroll to the bottom of the page to load additional pages. Ensure that you have bookmarked more than eight images using the bookmarklet, because that's the number of images you are displaying per page.

You can use your browser developer tools to track the AJAX requests. Usually, you can right-click anywhere on the website to open the contextual menu and click on **Inspect** or **Inspect Element** to access the web developer tools of your browser. Look for the panel for network requests. Reload the page and scroll to the bottom of the page to load new pages. You will see the request for the first page and the AJAX requests for additional pages, like in *Figure 6.20*:

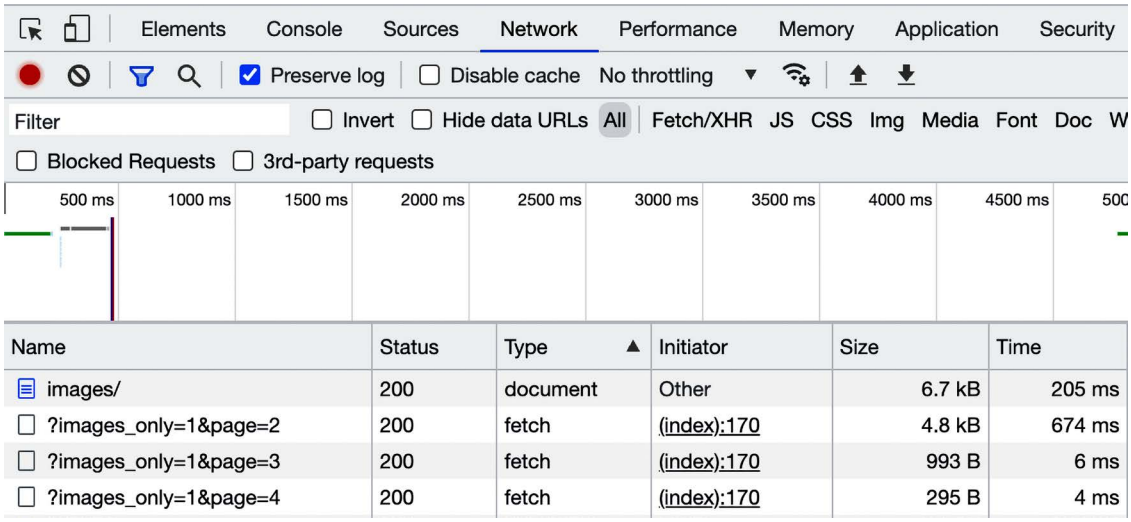


Figure 6.20: HTTP requests registered in the developer tools of the browser

In the shell where you are running Django, you will see the requests as well like this:

```
[08/Aug/2022 08:14:20] "GET /images/ HTTP/1.1" 200
[08/Aug/2022 08:14:25] "GET /images/?images_only=1&page=2 HTTP/1.1" 200
[08/Aug/2022 08:14:26] "GET /images/?images_only=1&page=3 HTTP/1.1" 200
[08/Aug/2022 08:14:26] "GET /images/?images_only=1&page=4 HTTP/1.1" 200
```

Finally, edit the `base.html` template of the `account` application and add the URL for the `images` item highlighted in bold:

```
<ul class="menu">
  ...
  <li {% if section == "images" %}class="selected"{% endif %}>
    <a href="{% url "images:list" %}">Images</a>
  </li>
  ...
</ul>
```

Now you can access the image list from the main menu.

Additional resources

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

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter06`
- Database indexes – `https://docs.djangoproject.com/en/4.1/ref/models/options/#django.db.models.Options.indexes`
- Many-to-many relationships – `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/`
- Requests HTTP library for Python – `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/`
- Pinterest browser button – `https://about.pinterest.com/en/browser-button`
- Static content for the account application – `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter06/bookmarks/images/static`
- CSS selectors – `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors`
- Locate DOM elements using CSS selectors – `https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Locating_DOM_elements_using_selectors`
- *Let's Encrypt* free automated certificate authority – `https://letsencrypt.org`
- Django `easy-thumbnails` app – `https://easy-thumbnails.readthedocs.io/`
- JavaScript Fetch API usage – `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch`
- JavaScript Cookie library – `https://github.com/js-cookie/js-cookie`
- Django's CSRF protection and AJAX – `https://docs.djangoproject.com/en/4.1/ref/csrf/#ajax`
- JavaScript Fetch API Request mode – `https://developer.mozilla.org/en-US/docs/Web/API/Request/mode`
- JavaScript Fetch API Response – `https://developer.mozilla.org/en-US/docs/Web/API/Response`

Summary

In this chapter, you created models with many-to-many relationships and learned how to customize the behavior of forms. You built a JavaScript bookmarklet to share images from other websites on your site. This chapter has also covered the creation of image thumbnails using the `easy-thumbnails` application. Finally, you implemented AJAX views using the JavaScript Fetch API and added infinite scroll pagination to the image list view.

In the next chapter, you will learn how to build a follow system and an activity stream. You will work with generic relations, signals, and denormalization. You will also learn how to use Redis with Django to count image views and generate an image ranking.

7

Tracking User Actions

In the previous chapter, you built a JavaScript bookmarklet to share content from other websites on your platform. You also implemented asynchronous actions with JavaScript in your project and created an infinite scroll.

In this chapter, you will learn how to build a follow system and create a user activity stream. You will also discover how Django signals work and integrate Redis's fast I/O storage into your project to store item views.

This chapter will cover the following points:

- Building a follow system
- Creating many-to-many relationships with an intermediary model
- Creating an activity stream application
- Adding generic relations to models
- Optimizing QuerySets for related objects
- Using signals for denormalizing counts
- Using Django Debug Toolbar to obtain relevant debug information
- Counting image views with Redis
- Creating a ranking of the most viewed images with Redis

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter07`.

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

Building a follow system

Let's build a follow system in your project. This means that your users will be able to follow each other and track what other users share on the platform. The relationship between users is a many-to-many relationship: a user can follow multiple users and they, in turn, can be followed by multiple users.

Creating many-to-many relationships with an intermediary model

In previous chapters, you created many-to-many relationships by adding the `ManyToManyField` to one of the related models and letting Django create the database table for the relationship. This is suitable for most cases, but sometimes you may need to create an intermediary model for the relationship. Creating an intermediary model is necessary when you want to store additional information about the relationship, for example, the date when the relationship was created, or a field that describes the nature of the relationship.

Let's create an intermediary model to build relationships between users. There are two reasons for using an intermediary model:

- You are using the `User` model provided by Django and you want to avoid altering it
- You want to store the time when the relationship was created

Edit the `models.py` file of the `account` application and add the following code to it:

```
class Contact(models.Model):
    user_from = models.ForeignKey('auth.User',
                                  related_name='rel_from_set',
                                  on_delete=models.CASCADE)
    user_to = models.ForeignKey('auth.User',
                                related_name='rel_to_set',
                                on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True)

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

    def __str__(self):
        return f'{self.user_from} follows {self.user_to}'
```

The preceding code shows the `Contact` model that you will use for user relationships. It contains the following fields:

- `user_from`: A `ForeignKey` for the user who creates the relationship
- `user_to`: A `ForeignKey` for the user being followed
- `created`: A `DateTimeField` field with `auto_now_add=True` to store the time when the relationship was created

A database index is automatically created on the `ForeignKey` fields. In the `Meta` class of the model, we have defined a database index in descending order for the `created` field. We have also added the `ordering` attribute to tell Django that it should sort results by the `created` field by default. We indicate descending order by using a hyphen before the field name, like `-created`.

Using the ORM, you could create a relationship for a user, `user1`, following another user, `user2`, like this:

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

The related managers, `rel_from_set` and `rel_to_set`, will return a QuerySet for the `Contact` model. In order to access the end side of the relationship from the `User` model, it would be desirable for `User` to contain a `ManyToManyField`, as follows:

```
following = models.ManyToManyField('self',
                                   through=Contact,
                                   related_name='followers',
                                   symmetrical=False)
```

In the preceding example, you tell Django to use your custom intermediary model for the relationship by adding `through=Contact` to the `ManyToManyField`. This is a many-to-many relationship from the `User` model to itself; you refer to `'self'` in the `ManyToManyField` field to create a relationship to the same model.

When you need additional fields in a many-to-many relationship, create a custom model with a `ForeignKey` for each side of the relationship. Add a `ManyToManyField` in one of the related models and indicate to Django that your intermediary model should be used by including it in the `through` parameter.

If the `User` model was part of your application, you could add the previous field to the model. However, you can't alter the `User` class directly because it belongs to the `django.contrib.auth` application. Let's take a slightly different approach by adding this field dynamically to the `User` model.

Edit the `models.py` file of the `account` application and add the following lines highlighted in bold:

```
from django.contrib.auth import get_user_model

# ...

# Add following field to User dynamically
user_model = get_user_model()
user_model.add_to_class('following',
                        models.ManyToManyField('self',
                            through=Contact,
                            related_name='followers',
                            symmetrical=False))
```

In the preceding code, you retrieve the user model by using the generic function `get_user_model()`, which is provided by Django. You use the `add_to_class()` method of Django models to monkey patch the `User` model.

Be aware that using `add_to_class()` is not the recommended way of adding fields to models. However, you take advantage of using it in this case to avoid creating a custom user model, keeping all the advantages of Django's built-in `User` model.

You also simplify the way that you retrieve related objects using the Django ORM with `user.followers.all()` and `user.following.all()`. You use the intermediary `Contact` model and avoid complex queries that would involve additional database joins, as would have been the case had you defined the relationship in your custom `Profile` model. The table for this many-to-many relationship will be created using the `Contact` model. Thus, the `ManyToManyField`, added dynamically, will not imply any database changes for the Django `User` model.

Keep in mind that, in most cases, it is preferable to add fields to the `Profile` model you created before, instead of monkey patching the `User` model. Ideally, you shouldn't alter the existing Django `User` model. Django allows you to use custom user models. If you want to use a custom user model, take a look at the documentation at `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#specifying-a-custom-user-model`.

Note that the relationship includes `symmetrical=False`. When you define a `ManyToManyField` in the model creating a relationship with itself, Django forces the relationship to be symmetrical. In this case, you are setting `symmetrical=False` to define a non-symmetrical relationship (if I follow you, it doesn't mean that you automatically follow me).

When you use an intermediary model for many-to-many relationships, some of the related manager's methods are disabled, such as `add()`, `create()`, or `remove()`. You need to create or delete instances of the intermediary model instead.

Run the following command to generate the initial migrations for the `account` application:

```
python manage.py makemigrations account
```

You will obtain an output like the following one:

```
Migrations for 'account':
  account/migrations/0002_auto_20220124_1106.py
    - Create model Contact
    - Create index account_con_created_8bdae6_idx on field(s) -created of model
contact
```

Now, run the following command to sync the application with the database:

```
python manage.py migrate account
```

You should see an output that includes the following line:

```
Applying account.0002_auto_20220124_1106... OK
```

The `Contact` model is now synced to the database, and you are able to create relationships between users. However, your site doesn't offer a way to browse users or see a particular user's profile yet. Let's build list and detail views for the `User` model.

Creating list and detail views for user profiles

Open the `views.py` file of the `account` application and add the following code highlighted in bold:

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User

# ...

@login_required
def user_list(request):
    users = User.objects.filter(is_active=True)
    return render(request,
                  'account/user/list.html',
                  {'section': 'people',
                   'users': users})

@login_required
def user_detail(request, username):
    user = get_object_or_404(User,
                             username=username,
                             is_active=True)
    return render(request,
                  'account/user/detail.html',
                  {'section': 'people',
                   'user': user})
```

These are simple list and detail views for `User` objects. The `user_list` view gets all active users. The Django `User` model contains an `is_active` flag to designate whether the user account is considered active. You filter the query by `is_active=True` to return only active users. This view returns all results, but you can improve it by adding pagination in the same way as you did for the `image_list` view.

The `user_detail` view uses the `get_object_or_404()` shortcut to retrieve the active user with the given username. The view returns an HTTP `404` response if no active user with the given username is found.

Edit the `urls.py` file of the `account` application, and add a URL pattern for each view, as follows. New code is highlighted in bold:

```
urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
```

```
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
    path('users/', views.user_list, name='user_list'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

You will use the `user_detail` URL pattern to generate the canonical URL for users. You have already defined a `get_absolute_url()` method in a model to return the canonical URL for each object. Another way to specify the URL for a model is by adding the `ABSOLUTE_URL_OVERRIDES` setting to your project.

Edit the `settings.py` file of your project and add the following code highlighted in bold:

```
from django.urls import reverse_lazy

# ...

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])
}
```

Django adds a `get_absolute_url()` method dynamically to any models that appear in the `ABSOLUTE_URL_OVERRIDES` setting. This method returns the corresponding URL for the given model specified in the setting. You return the `user_detail` URL for the given user. Now, you can use `get_absolute_url()` on a `User` instance to retrieve its corresponding URL.

Open the Python shell with the following command:

```
python manage.py shell
```

Then run the following code to test it:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.latest('id')
>>> str(user.get_absolute_url())
'/account/users/ellington/'
```

The returned URL follows the expected format `/account/users/<username>/`.

You will need to create templates for the views that you just built. Add the following directory and files to the `templates/account/` directory of the `account` application:

```
/user/
    detail.html
    list.html
```

Edit the `account/user/list.html` template and add the following code to it:

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

{% block title %}People{% endblock %}

{% block content %}
  <h1>People</h1>
  <div id="people-list">
    {% for user in users %}
      <div class="user">
        <a href="{{ user.get_absolute_url }}">
          <img src="{% thumbnail user.profile.photo 180x180 %}">
        </a>
        <div class="info">
          <a href="{{ user.get_absolute_url }}" class="title">
            {{ user.get_full_name }}
          </a>
        </div>
      </div>
    {% endfor %}
  </div>
{% endblock %}
```

The preceding template allows you to list all the active users on the site. You iterate over the given users and use the `{% thumbnail %}` template tag from `easy-thumbnails` to generate profile image thumbnails.

Note that the users need to have a profile image. To use a default image for users that don't have a profile image, you can add an `if/else` statement to check whether the user has a profile photo, like `{% if user.profile.photo %} {# photo thumbnail #} {% else %} {# default image #} {% endif %}`.

Open the `base.html` template of your project and include the `user_list` URL in the `href` attribute of the following menu item. New code is highlighted in bold:

```
<ul class="menu">
  ...
  <li {% if section == "people" %}class="selected"{% endif %}>
    <a href="{% url "user_list" %}">People</a>
  </li>
</ul>
```

Start the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/account/users/` in your browser. You should see a list of users like the following one:

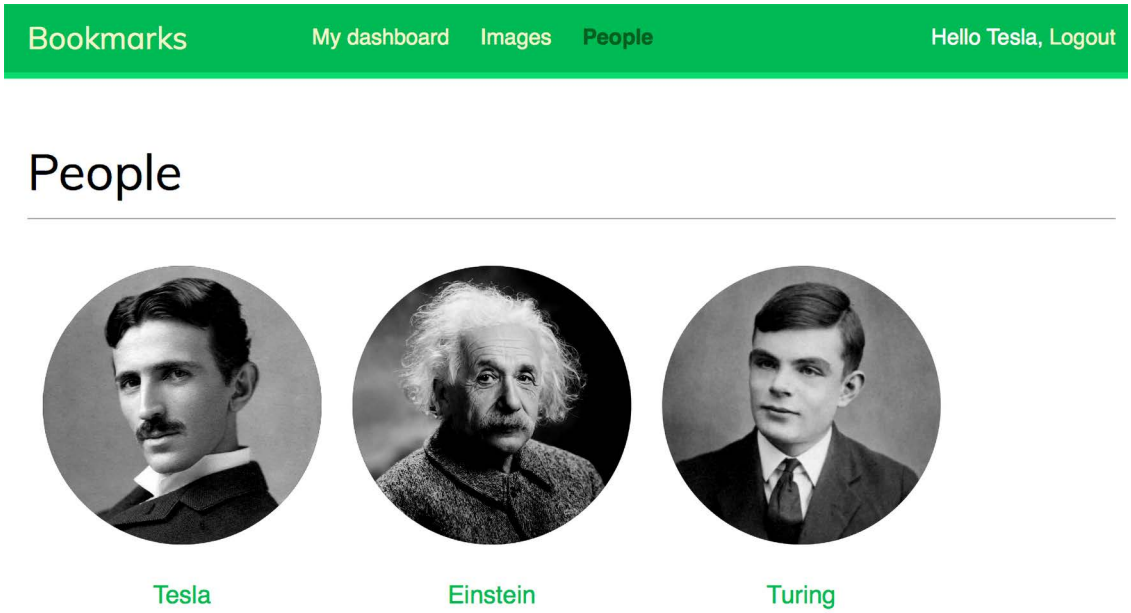


Figure 7.1: The user list page with profile image thumbnails

Remember that if you have any difficulty generating thumbnails, you can add `THUMBNAIL_DEBUG = True` to your `settings.py` file in order to obtain debug information in the shell.

Edit the `account/user/detail.html` template of the `account` application and add the following code to it:

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

{% block title %}{{ user.get_full_name }}{% endblock %}

{% block content %}
  <h1>{{ user.get_full_name }}</h1>
  <div class="profile-info">
    <img src="{% thumbnail user.profile.photo 180x180 %}" class="user-detail">
  </div>
  {% with total_followers=user.followers.count %}
    <span class="count">
      <span class="total">{{ total_followers }}</span>
      follower{{ total_followers|pluralize }}
    </span>
    <a href="#" data-id="{{ user.id }}" data-action="{% if request.user in
user.followers.all %}un{% endif %}follow" class="follow button">
```

```
      {% if request.user not in user.followers.all %}
        Follow
      {% else %}
        Unfollow
      {% endif %}
    </a>
    <div id="image-list" class="image-container">
      {% include "images/image/list_images.html" with images=user.images_
created.all %}
    </div>
  {% endwith %}
{% endblock %}
```

Make sure that no template tag is split onto multiple lines; Django doesn't support multiple-line tags.

In the `detail` template, the user profile is displayed and the `{% thumbnail %}` template tag is used to show the profile image. The total number of followers is presented and a link to follow or unfollow the user. This link will be used to follow/unfollow a particular user. The `data-id` and `data-action` attributes of the `<a>` HTML element contain the user ID and the initial action to perform when the link element is clicked – `follow` or `unfollow`. The initial action (*follow* or *unfollow*) depends on whether the user requesting the page is already a follower of the user. The images bookmarked by the user are displayed by including the `images/image/list_images.html` template.

Open your browser again and click on a user who has bookmarked some images. The user page will look as follows:

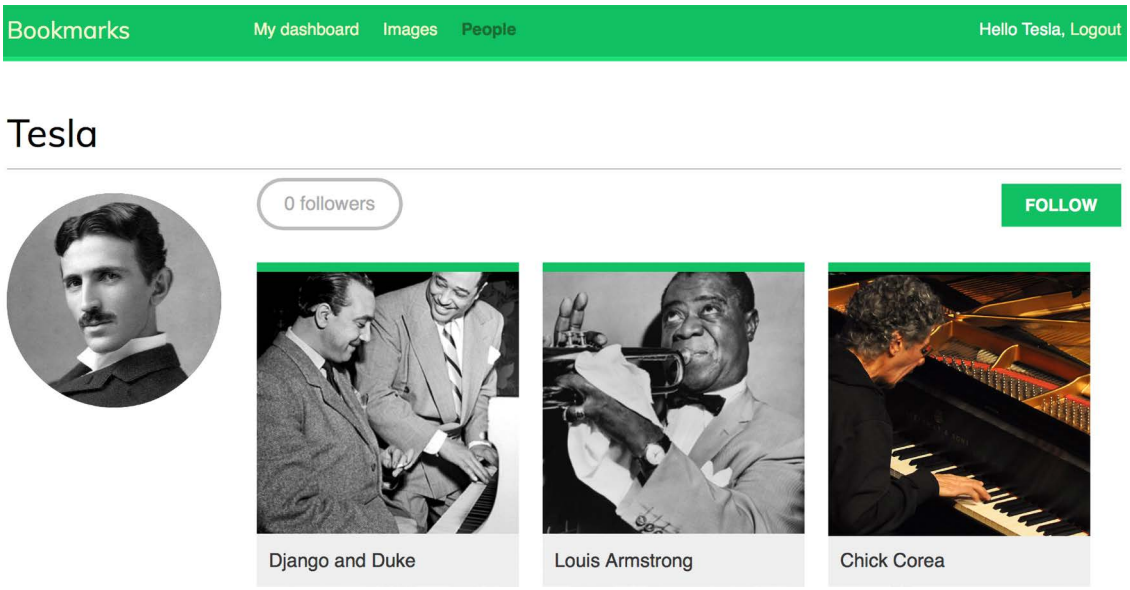


Figure 7.2: The user detail page

Image of *Chick Corea* by ataelw (license: Creative Commons Attribution 2.0 Generic: `https://creativecommons.org/licenses/by/2.0/`)

Adding user follow/unfollow actions with JavaScript

Let's add functionality to follow/unfollow users. We will create a new view to follow/unfollow users and implement an asynchronous HTTP request with JavaScript for the follow/unfollow action.

Edit the `views.py` file of the `account` application and add the following code highlighted in bold:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from .models import Contact

# ...

@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                    user_from=request.user,
                    user_to=user)
            else:
                Contact.objects.filter(user_from=request.user,
                                       user_to=user).delete()
            return JsonResponse({'status':'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status':'error'})
    return JsonResponse({'status':'error'})
```

The `user_follow` view is quite similar to the `image_like` view that you created in *Chapter 6*, *Sharing Content on Your Website*. Since you are using a custom intermediary model for the user's many-to-many relationship, the default `add()` and `remove()` methods of the automatic manager of `ManyToManyField` are not available. Instead, the intermediary `Contact` model is used to create or delete user relationships.

Edit the `urls.py` file of the `account` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
    path('users/', views.user_list, name='user_list'),
    path('users/follow/', views.user_follow, name='user_follow'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

Ensure that you place the preceding pattern before the `user_detail` URL pattern. Otherwise, any requests to `/users/follow/` will match the regular expression of the `user_detail` pattern and that view will be executed instead. Remember that in every HTTP request, Django checks the requested URL against each pattern in order of appearance and stops at the first match.

Edit the `user/detail.html` template of the `account` application and append the following code to it:

```
{% block domready %}
  var const = '{% url "user_follow" %}';
  var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrftoken},
    mode: 'same-origin'
  }

  document.querySelector('a.follow')
          .addEventListener('click', function(e){
    e.preventDefault();
    var followButton = this;

    // add request body
    var formData = new FormData();
    formData.append('id', followButton.dataset.id);
    formData.append('action', followButton.dataset.action);
    options['body'] = formData;

    // send HTTP request
    fetch(url, options)
    .then(response => response.json())
    .then(data => {
      if (data['status'] === 'ok')
```

```
      {
        var previousAction = followButton.dataset.action;

        // toggle button text and data-action
        var action = previousAction === 'follow' ? 'unfollow' : 'follow';
        followButton.dataset.action = action;
        followButton.innerHTML = action;

        // update follower count
        var followerCount = document.querySelector('span.count .total');
        var totalFollowers = parseInt(followerCount.innerHTML);
        followerCount.innerHTML = previousAction === 'follow' ? totalFollowers
+ 1 : totalFollowers - 1;
      }
    })
  });
{% endblock %}
```

The preceding template block contains the JavaScript code to perform the asynchronous HTTP request to follow or unfollow a particular user and also to toggle the follow/unfollow link. The Fetch API is used to perform the AJAX request and set both the `data-action` attribute and the text of the HTML `<a>` element based on its previous value. When the action is completed, the total number of followers displayed on the page is updated as well.

Open the user detail page of an existing user and click on the **FOLLOW** link to test the functionality you just built. You will see that the followers count is increased:

Figure 7.3: The followers count and follow/unfollow button

The follow system is now complete, and users can follow each other. Next, we will build an activity stream creating relevant content for each user that is based on the people they follow.

Building a generic activity stream application

Many social websites display an activity stream to their users so that they can track what other users do on the platform. An activity stream is a list of recent activities performed by a user or a group of users. For example, Facebook's News Feed is an activity stream. Sample actions can be *user X bookmarked image Y* or *user X is now following user Y*.

You are going to build an activity stream application so that every user can see the recent interactions of the users they follow. To do so, you will need a model to save the actions performed by users on the website and a simple way to add actions to the feed.

Create a new application named `actions` inside your project with the following command:

```
python manage.py startapp actions
```

Add the new application to `INSTALLED_APPS` in the `settings.py` file of your project to activate the application in your project. The new line is highlighted in bold:

```
INSTALLED_APPS = [
    # ...
    'actions.apps.ActionsConfig',
]
```

Edit the `models.py` file of the `actions` application and add the following code to it:

```
from django.db import models

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                             related_name='actions',
                             on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True)

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

The preceding code shows the `Action` model that will be used to store user activities. The fields of this model are as follows:

- `user`: The user who performed the action; this is a `ForeignKey` to the Django `User` model.
- `verb`: The verb describing the action that the user has performed.
- `created`: The date and time when this action was created. We use `auto_now_add=True` to automatically set this to the current datetime when the object is saved for the first time in the database.

In the `Meta` class of the model, we have defined a database index in descending order for the `created` field. We have also added the `ordering` attribute to tell Django that it should sort results by the `created` field in descending order by default.

With this basic model, you can only store actions such as *user X did something*. You need an extra `ForeignKey` field to save actions that involve a `target` object, such as *user X bookmarked image Y* or *user X is now following user Y*. As you already know, a normal `ForeignKey` can point to only one model. Instead, you will need a way for the action's `target` object to be an instance of an existing model. This is what the Django `contenttypes` framework will help you to do.

Using the contenttypes framework

Django includes a `contenttypes` framework located at `django.contrib.contenttypes`. This application can track all models installed in your project and provides a generic interface to interact with your models.

The `django.contrib.contenttypes` application is included in the `INSTALLED_APPS` setting by default when you create a new project using the `startproject` command. It is used by other `contrib` packages, such as the authentication framework and the administration application.

The `contenttypes` application contains a `ContentType` model. Instances of this model represent the actual models of your application, and new instances of `ContentType` are automatically created when new models are installed in your project. The `ContentType` model has the following fields:

- `app_label`: This indicates the name of the application that the model belongs to. This is automatically taken from the `app_label` attribute of the model `Meta` options. For example, your `Image` model belongs to the `images` application.
- `model`: The name of the model class.
- `name`: This indicates the human-readable name of the model. This is automatically taken from the `verbose_name` attribute of the model `Meta` options.

Let's take a look at how you can interact with `ContentType` objects. Open the shell using the following command:

```
python manage.py shell
```

You can obtain the `ContentType` object corresponding to a specific model by performing a query with the `app_label` and `model` attributes, as follows:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> image_type = ContentType.objects.get(app_label='images', model='image')
>>> image_type
<ContentType: images | image>
```

You can also retrieve the model class from a `ContentType` object by calling its `model_class()` method:

```
>>> image_type.model_class()
<class 'images.models.Image'>
```

It's also common to obtain the `ContentType` object for a particular model class, as follows:

```
>>> from images.models import Image
>>> ContentType.objects.get_for_model(Image)
<ContentType: images | image>
```

These are just some examples of using `contenttypes`. Django offers more ways to work with them. You can find the official documentation for the `contenttypes` framework at `https://docs.djangoproject.com/en/4.1/ref/contrib/contenttypes/`.

Adding generic relations to your models

In generic relations, `ContentType` objects play the role of pointing to the model used for the relationship. You will need three fields to set up a generic relation in a model:

- A `ForeignKey` field to `ContentType`: This will tell you the model for the relationship
- A field to store the primary key of the related object: This will usually be a `PositiveIntegerField` to match Django's automatic primary key fields
- A field to define and manage the generic relation using the two previous fields: The `contenttypes` framework offers a `GenericForeignKey` field for this purpose

Edit the `models.py` file of the `actions` application and add the following code highlighted in bold:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                             related_name='actions',
                             on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
   created = models.DateTimeField(auto_now_add=True)
    target_ct = models.ForeignKey(ContentType,
                                  blank=True,
                                  null=True,
                                  related_name='target_obj',
                                  on_delete=models.CASCADE)
    target_id = models.PositiveIntegerField(null=True,
                                            blank=True)
    target = GenericForeignKey('target_ct', 'target_id')

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

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

- `target_ct`: A `ForeignKey` field that points to the `ContentType` model
- `target_id`: A `PositiveIntegerField` for storing the primary key of the related object
- `target`: A `GenericForeignKey` field to the related object based on the combination of the two previous fields

We have also added a multiple-field index including the `target_ct` and `target_id` fields.

Django does not create `GenericForeignKey` fields in the database. The only fields that are mapped to database fields are `target_ct` and `target_id`. Both fields have `blank=True` and `null=True` attributes, so that a `target` object is not required when saving `Action` objects.

You can make your applications more flexible by using generic relations instead of foreign keys.

Run the following command to create initial migrations for this application:

```
python manage.py makemigrations actions
```

You should see the following output:

```
Migrations for 'actions':
  actions/migrations/0001_initial.py
    - Create model Action
    - Create index actions_act_created_64f10d_idx on field(s) -created of model
action
    - Create index actions_act_target__f20513_idx on field(s) target_ct,
target_id of model action
```

Then, run the next command to sync the application with the database:

```
python manage.py migrate
```

The output of the command should indicate that the new migrations have been applied, as follows:

```
Applying actions.0001_initial... OK
```

Let's add the `Action` model to the administration site. Edit the `admin.py` file of the `actions` application and add the following code to it:

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

@admin.register(Action)
class ActionAdmin(admin.ModelAdmin):
```

```
    list_display = ['user', 'verb', 'target', 'created']
    list_filter = ['created']
    search_fields = ['verb']
```

You just registered the `Action` model on the administration site.

Start the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/actions/action/add/` in your browser. You should see the page for creating a new `Action` object, as follows:

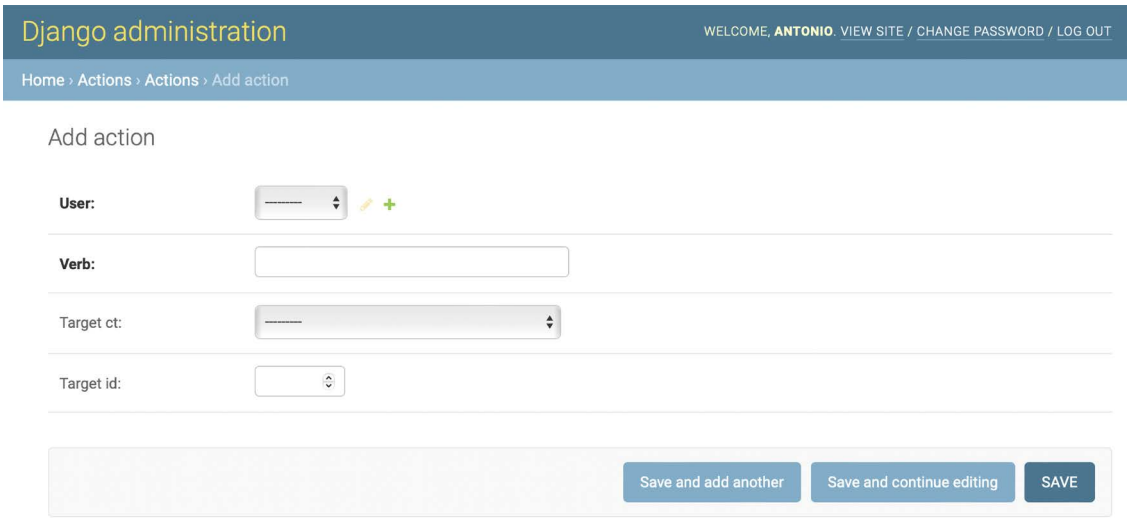


Figure 7.4: The Add action page on the Django administration site

As you will notice in the preceding screenshot, only the `target_ct` and `target_id` fields that are mapped to actual database fields are shown. The `GenericForeignKey` field does not appear in the form. The `target_ct` field allows you to select any of the registered models of your Django project. You can restrict the content types to choose from a limited set of models using the `limit_choices_to` attribute in the `target_ct` field; the `limit_choices_to` attribute allows you to restrict the content of `ForeignKey` fields to a specific set of values.

Create a new file inside the `actions` application directory and name it `utils.py`. You need to define a shortcut function that will allow you to create new `Action` objects in a simple way. Edit the new `utils.py` file and add the following code to it:

```
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    action = Action(user=user, verb=verb, target=target)
    action.save()
```

The `create_action()` function allows you to create actions that optionally include a `target` object. You can use this function anywhere in your code as a shortcut to add new actions to the activity stream.

Avoiding duplicate actions in the activity stream

Sometimes, your users might click several times on the **Like** or **Unlike** button or perform the same action multiple times in a short period of time. This will easily lead to storing and displaying duplicate actions. To avoid this, let's improve the `create_action()` function to skip obvious duplicated actions.

Edit the `utils.py` file of the `actions` application, as follows:

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    # check for any similar action made in the last minute
    now = timezone.now()
    last_minute = now - datetime.timedelta(seconds=60)
    similar_actions = Action.objects.filter(user_id=user.id,
                                            verb= verb,
                                            created__gte=last_minute)
    if target:
        target_ct = ContentType.objects.get_for_model(target)
        similar_actions = similar_actions.filter(
                                                target_ct=target_ct,
                                                target_id=target.id)
    if not similar_actions:
        # no existing actions found
        action = Action(user=user, verb=verb, target=target)
        action.save()
        return True
    return False
```

You have changed the `create_action()` function to avoid saving duplicate actions and return a Boolean to tell you whether the action was saved. This is how you avoid duplicates:

1. First, you get the current time using the `timezone.now()` method provided by Django. This method does the same as `datetime.datetime.now()` but returns a timezone-aware object. Django provides a setting called `USE_TZ` to enable or disable timezone support. The default `settings.py` file created using the `startproject` command includes `USE_TZ=True`.
2. You use the `last_minute` variable to store the datetime from one minute ago and retrieve any identical actions performed by the user since then.

3. You create an `Action` object if no identical action already exists in the last minute. You return `True` if an `Action` object was created, or `False` otherwise.

Adding user actions to the activity stream

It's time to add some actions to your views to build the activity stream for your users. You will store an action for each of the following interactions:

- A user bookmarks an image
- A user likes an image
- A user creates an account
- A user starts following another user

Edit the `views.py` file of the `images` application and add the following import:

```
from actions.utils import create_action
```

In the `image_create` view, add `create_action()` after saving the image, like this. The new line is highlighted in bold:

```
@login_required
def image_create(request):
    if request.method == 'POST':
        # form is sent
        form = ImageCreateForm(data=request.POST)
        if form.is_valid():
            # form data is valid
            cd = form.cleaned_data
            new_image = form.save(commit=False)
            # assign current user to the item
            new_image.user = request.user
            new_image.save()
            create_action(request.user, 'bookmarked image', new_image)
            messages.success(request, 'Image added successfully')
            # redirect to new created image detail view
            return redirect(new_image.get_absolute_url())
    else:
        # build form with data provided by the bookmarklet via GET
        form = ImageCreateForm(data=request.GET)
    return render(request,
                  'images/image/create.html',
                  {'section': 'images',
                   'form': form})
```

In the `image_like` view, add `create_action()` after adding the user to the `users_like` relationship, as follows. The new line is highlighted in bold:

```
@login_required
@require_POST
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
            if action == 'like':
                image.users_like.add(request.user)
                create_action(request.user, 'likes', image)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status':'ok'})
        except Image.DoesNotExist:
            pass
    return JsonResponse({'status':'error'})
```

Now, edit the `views.py` file of the `account` application and add the following import:

```
from actions.utils import create_action
```

In the `register` view, add `create_action()` after creating the `Profile` object, as follows. The new line is highlighted in bold:

```
def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Create a new user object but avoid saving it yet
            new_user = user_form.save(commit=False)
            # Set the chosen password
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Save the User object
            new_user.save()
            # Create the user profile
            Profile.objects.create(user=new_user)
            create_action(new_user, 'has created an account')
            return render(request,
```

```
                            'account/register_done.html',
                            {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

In the `user_follow` view, add `create_action()` as follows. The new line is highlighted in bold:

```
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                    user_from=request.user,
                    user_to=user)
                create_action(request.user, 'is following', user)
            else:
                Contact.objects.filter(user_from=request.user,
                                       user_to=user).delete()
            return JsonResponse({'status':'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status':'error'})
    return JsonResponse({'status':'error'})
```

As you can see in the preceding code, thanks to the `Action` model and the helper function, it's very easy to save new actions to the activity stream.

Displaying the activity stream

Finally, you need a way to display the activity stream for each user. You will include the activity stream on the user's dashboard. Edit the `views.py` file of the `account` application. Import the `Action` model and modify the `dashboard` view, as follows. New code is highlighted in bold:

```
from actions.models import Action

# ...
```

```
@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # If user is following others, retrieve only their actions
        actions = actions.filter(user_id__in=following_ids)
    actions = actions[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

In the preceding view, you retrieve all actions from the database, excluding the ones performed by the current user. By default, you retrieve the latest actions performed by all users on the platform. If the user is following other users, you restrict the query to retrieve only the actions performed by the users they follow. Finally, you limit the result to the first 10 actions returned. You don't use `order_by()` in the QuerySet because you rely on the default ordering that you provided in the `Meta` options of the `Action` model. Recent actions will come first since you set `ordering = ['-created']` in the `Action` model.

Optimizing QuerySets that involve related objects

Every time you retrieve an `Action` object, you will usually access its related `User` object and the user's related `Profile` object. The Django ORM offers a simple way to retrieve related objects at the same time, thereby avoiding additional queries to the database.

Using select_related()

Django offers a QuerySet method called `select_related()` that allows you to retrieve related objects for one-to-many relationships. This translates to a single, more complex QuerySet, but you avoid additional queries when accessing the related objects. The `select_related` method is for `ForeignKey` and `OneToOne` fields. It works by performing a SQL `JOIN` and including the fields of the related object in the `SELECT` statement.

To take advantage of `select_related()`, edit the following line of the preceding code in the `views.py` file of the account application to add `select_related`, including the fields that you will use, like this. Edit the `views.py` file of the account application. New code is highlighted in bold:

```
@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
```

```
    if following_ids:
        # If user is following others, retrieve only their actions
        actions = actions.filter(user_id__in=following_ids)
    actions = actions.select_related('user', 'user__profile')[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

You use `user__profile` to join the `Profile` table in a single SQL query. If you call `select_related()` without passing any arguments to it, it will retrieve objects from all `ForeignKey` relationships. Always limit `select_related()` to the relationships that will be accessed afterward.

Using `select_related()` carefully can vastly improve execution time.

Using prefetch_related()

`select_related()` will help you boost the performance for retrieving related objects in one-to-many relationships. However, `select_related()` doesn't work for many-to-many or many-to-one relationships (`ManyToMany` or reverse `ForeignKey` fields). Django offers a different QuerySet method called `prefetch_related` that works for many-to-many and many-to-one relationships in addition to the relationships supported by `select_related()`. The `prefetch_related()` method performs a separate lookup for each relationship and joins the results using Python. This method also supports the prefetching of `GenericRelation` and `GenericForeignKey`.

Edit the `views.py` file of the `account` application and complete your query by adding `prefetch_related()` to it for the target `GenericForeignKey` field, as follows. The new code is highlighted in bold:

```
@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # If user is following others, retrieve only their actions
        actions = actions.filter(user_id__in=following_ids)
    actions = actions.select_related('user', 'user__profile')\
                     .prefetch_related('target')[:10]
    return render(request,
                  'account/dashboard.html',
```

```
                  {'section': 'dashboard',
                   'actions': actions})

actions = actions.select_related('user', 'user__profile'
```

This query is now optimized for retrieving the user actions, including related objects.

Creating templates for actions

Let's now create the template to display a particular `Action` object. Create a new directory inside the `actions` application directory and name it `templates`. Add the following file structure to it:

```
actions/
    action/
        detail.html
```

Edit the `actions/action/detail.html` template file and add the following lines to it:

```
{% load thumbnail %}

{% with user=action.user profile=action.user.profile %}
<div class="action">
  <div class="images">
    {% if profile.photo %}
      {% thumbnail user.profile.photo "80x80" crop="100%" as im %}
      <a href="{{ user.get_absolute_url }}">
        <img src="{{ im.url }}" alt="{{ user.get_full_name }}"
         class="item-img">
      </a>
    {% endif %}
    {% if action.target %}
      {% with target=action.target %}
        {% if target.image %}
          {% thumbnail target.image "80x80" crop="100%" as im %}
          <a href="{{ target.get_absolute_url }}">
            <img src="{{ im.url }}" class="item-img">
          </a>
        {% endif %}
      {% endwith %}
    {% endif %}
  </div>
  <div class="info">
    <p>
      <span class="date">{{ action.created|timesince }} ago</span>
```

```
      <br />
      <a href="{{ user.get_absolute_url }}">
        {{ user.first_name }}
      </a>
      {{ action.verb }}
      {% if action.target %}
        {% with target=action.target %}
          <a href="{{ target.get_absolute_url }}">{{ target }}</a>
        {% endwith %}
      {% endif %}
    </p>
  </div>
</div>
{% endwith %}
```

This is the template used to display an `Action` object. First, you use the `{% with %}` template tag to retrieve the user performing the action and the related `Profile` object. Then, you display the image of the `target` object if the `Action` object has a related `target` object. Finally, you display the link to the user who performed the action, the verb, and the `target` object, if any.

Edit the `account/dashboard.html` template of the `account` application and append the following code highlighted in bold to the bottom of the `content` block:

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

{% block title %}Dashboard{% endblock %}

{% block content %}

  ...

  <h2>What's happening</h2>
  <div id="action-list">
    {% for action in actions %}
      {% include "actions/action/detail.html" %}
    {% endfor %}
  </div>
{% endblock %}
```

Open `http://127.0.0.1:8000/account/` in your browser. Log in as an existing user and perform several actions so that they get stored in the database. Then, log in using another user, follow the previous user, and take a look at the generated action stream on the dashboard page.

It should look like the following:

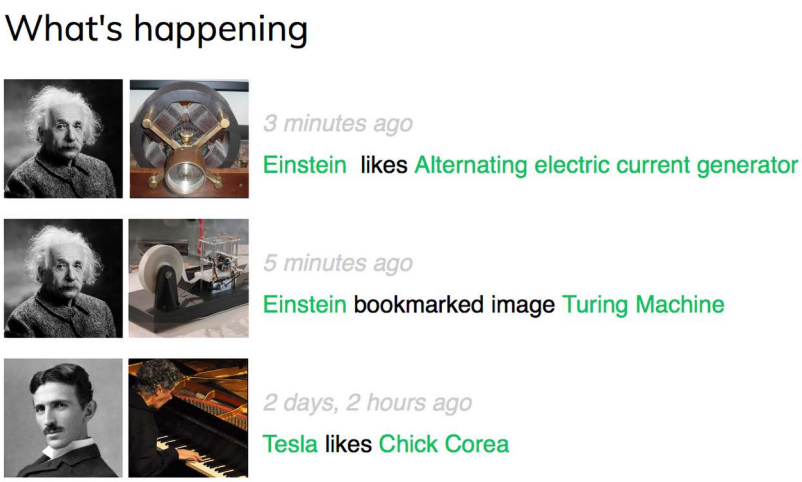


Figure 7.5: The activity stream for the current user

Figure 7.5 image attributions:

- *Tesla's induction motor by Ctac (license: Creative Commons Attribution Share-Alike 3.0 Unported:* `https://creativecommons.org/licenses/by-sa/3.0/`*)*
- *Turing Machine Model Davey 2012 by Rocky Acosta (license: Creative Commons Attribution 3.0 Unported:* `https://creativecommons.org/licenses/by/3.0/`*)*
- *Chick Corea* by ataelw (license: Creative Commons Attribution 2.0 Generic: `https://creativecommons.org/licenses/by/2.0/`)

You just created a complete activity stream for your users, and you can easily add new user actions to it. You can also add infinite scroll functionality to the activity stream by implementing the same AJAX paginator that you used for the `image_list` view. Next, you will learn how to use Django signals to denormalize action counts.

Using signals for denormalizing counts

There are some cases when you may want to denormalize your data. Denormalization is making data redundant in such a way that it optimizes read performance. For example, you might be copying related data to an object to avoid expensive read queries to the database when retrieving the related data. You have to be careful about denormalization and only start using it when you really need it. The biggest issue you will find with denormalization is that it's difficult to keep your denormalized data updated.

Let's take a look at an example of how to improve your queries by denormalizing counts. You will denormalize data from your `Image` model and use Django signals to keep the data updated.

Working with signals

Django comes with a signal dispatcher that allows receiver functions to get notified when certain actions occur. Signals are very useful when you need your code to do something every time something else happens. Signals allow you to decouple logic: you can capture a certain action, regardless of the application or code that triggered that action, and implement logic that gets executed whenever that action occurs. For example, you can build a signal receiver function that gets executed every time a `User` object is saved. You can also create your own signals so that others can get notified when an event happens.

Django provides several signals for models located at `django.db.models.signals`. Some of these signals are as follows:

- `pre_save` and `post_save` are sent before or after calling the `save()` method of a model
- `pre_delete` and `post_delete` are sent before or after calling the `delete()` method of a model or QuerySet
- `m2m_changed` is sent when a `ManyToManyField` on a model is changed

These are just a subset of the signals provided by Django. You can find a list of all built-in signals at `https://docs.djangoproject.com/en/4.1/ref/signals/`.

Let's say you want to retrieve images by popularity. You can use the Django aggregation functions to retrieve images ordered by the number of users who like them. Remember that you used Django aggregation functions in *Chapter 3*, *Extending Your Blog Application*. The following code example will retrieve images according to their number of likes:

```
from django.db.models import Count
from images.models import Image
images_by_popularity = Image.objects.annotate(
    total_likes=Count('users_like')).order_by('-total_likes')
```

However, ordering images by counting their total `likes` is more expensive in terms of performance than ordering them by a field that stores total counts. You can add a field to the `Image` model to denormalize the total number of likes to boost performance in queries that involve this field. The issue is how to keep this field updated.

Edit the `models.py` file of the `images` application and add the following `total_likes` field to the `Image` model. The new code is highlighted in bold:

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(default=0)

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

```
            models.Index(fields=['-total_likes']),
        ]
        ordering = ['-created']
```

The `total_likes` field will allow you to store the total count of users who like each image. Denormalizing counts is useful when you want to filter or order QuerySets by them. We have added a database index for the `total_likes` field in descending order because we plan to retrieve images ordered by their total likes in descending order.

There are several ways to improve performance that you have to take into account before denormalizing fields. Consider database indexes, query optimization, and caching before starting to denormalize your data.

Run the following command to create the migrations for adding the new field to the database table:

```
python manage.py makemigrations images
```

You should see the following output:

```
Migrations for 'images':
  images/migrations/0002_auto_20220124_1757.py
    - Add field total_likes to image
    - Create index images_imag_total_l_0bcd7e_idx on field(s) -total_likes of
model image
```

Then, run the following command to apply the migration:

```
python manage.py migrate images
```

The output should include the following line:

```
Applying images.0002_auto_20220124_1757... OK
```

You need to attach a `receiver` function to the `m2m_changed` signal.

Create a new file inside the `images` application directory and name it `signals.py`. Add the following code to it:

```
from django.db.models.signals import m2m_changed
from django.dispatch import receiver
from .models import Image

@receiver(m2m_changed, sender=Image.users_like.through)
def users_like_changed(sender, instance, **kwargs):
    instance.total_likes = instance.users_like.count()
    instance.save()
```

First, you register the `users_like_changed` function as a receiver function using the `receiver()` decorator. You attach it to the `m2m_changed` signal. Then, you connect the function to `Image.users_like.through` so that the function is only called if the `m2m_changed` signal has been launched by this sender. There is an alternate method for registering a receiver function; it consists of using the `connect()` method of the `Signal` object.

Django signals are synchronous and blocking. Don't confuse signals with asynchronous tasks. However, you can combine both to launch asynchronous tasks when your code gets notified by a signal. You will learn how to create asynchronous tasks with Celery in *Chapter 8*, *Building an Online Shop*.

You have to connect your receiver function to a signal so that it gets called every time the signal is sent. The recommended method for registering your signals is by importing them into the `ready()` method of your application configuration class. Django provides an application registry that allows you to configure and intropect your applications.

Application configuration classes

Django allows you to specify configuration classes for your applications. When you create an application using the `startapp` command, Django adds an `apps.py` file to the application directory, including a basic application configuration that inherits from the `AppConfig` class.

The application configuration class allows you to store metadata and the configuration for the application, and it provides introspection for the application. You can find more information about application configurations at `https://docs.djangoproject.com/en/4.1/ref/applications/`.

In order to register your signal `receiver` functions, when you use the `receiver()` decorator, you just need to import the `signals` module of your application inside the `ready()` method of the application configuration class. This method is called as soon as the application registry is fully populated. Any other initializations for your application should also be included in this method.

Edit the `apps.py` file of the `images` application and add the following code highlighted in bold:

```
from django.apps import AppConfig

class ImagesConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'images'

    def ready(self):
        # import signal handlers
        import images.signals
```

You import the signals for this application in the `ready()` method so that they are imported when the `images` application is loaded.

Run the development server with the following command:

```
python manage.py runserver
```

Open your browser to view an image detail page and click on the **Like** button.

Go to the administration site, navigate to the edit image URL, such as `http://127.0.0.1:8000/admin/images/image/1/change/`, and take a look at the `total_likes` attribute. You should see that the `total_likes` attribute is updated with the total number of users who like the image, as follows:

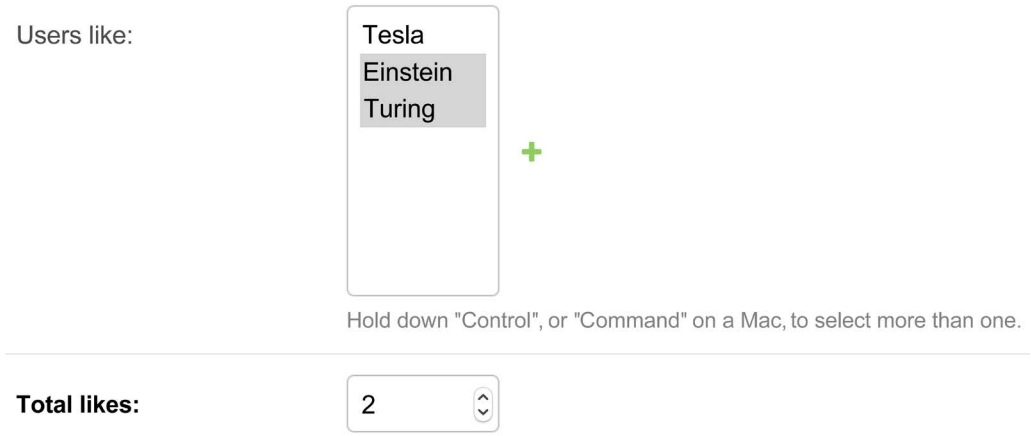


Figure 7.6: The image edit page on the administration site, including denormalization for total likes

Now, you can use the `total_likes` attribute to order images by popularity or display the value anywhere, avoiding using complex queries to calculate it.

Consider the following query to get images ordered by their likes count in descending order:

```
from django.db.models import Count
images_by_popularity = Image.objects.annotate(
    likes=Count('users_like')).order_by('-likes')
```

The preceding query can now be written as follows:

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

This results in a less expensive SQL query thanks to denormalizing the total likes for images. You have also learned how you can use Django signals.

Use signals with caution since they make it difficult to know the control flow. In many cases, you can avoid using signals if you know which receivers need to be notified.

You will need to set initial counts for the rest of the `Image` objects to match the current status of the database.

Open the shell with the following command:

```
python manage.py shell
```

Execute the following code in the shell:

```
>>> from images.models import Image
>>> for image in Image.objects.all():
...     image.total_likes = image.users_like.count()
...     image.save()
```

You have manually updated the likes count for the existing images in the database. From now on, the `users_like_changed` signal receiver function will handle updating the `total_likes` field whenever the many-to-many related objects change.

Next, you will learn how to use Django Debug Toolbar to obtain relevant debug information for requests, including execution time, SQL queries executed, templates rendered, signals registered, and much more.

Using Django Debug Toolbar

At this point, you will already be familiar with Django's debug page. Throughout the previous chapters, you have seen the distinctive yellow and grey Django debug page several times. For example, in *Chapter 2*, *Enhancing Your Blog with Advanced Features*, in the *Handling pagination errors* section, the debug page showed information related to unhandled exceptions when implementing object pagination.

The Django debug page provides useful debug information. However, there is a Django application that includes more detailed debug information and can be really helpful when developing.

Django Debug Toolbar is an external Django application that allows you to see relevant debug information about the current request/response cycle. The information is divided into multiple panels that show different information, including request/response data, Python package versions used, execution time, settings, headers, SQL queries, templates used, cache, signals, and logging.

You can find the documentation for Django Debug Toolbar at `https://django-debug-toolbar.readthedocs.io/`.

Installing Django Debug Toolbar

Install `django-debug-toolbar` via `pip` using the following command:

```
pip install django-debug-toolbar==3.6.0
```

Edit the `settings.py` file of your project and add `debug_toolbar` to the `INSTALLED_APPS` setting, as follows. The new line is highlighted in bold:

```
INSTALLED_APPS = [
    # ...
    'debug_toolbar',
]
```

In the same file, add the following line highlighted in bold to the `MIDDLEWARE` setting:

```
MIDDLEWARE = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Django Debug Toolbar is mostly implemented as middleware. The order of `MIDDLEWARE` is important. `DebugToolbarMiddleware` has to be placed before any other middleware, except for middleware that encodes the response's content, such as `GZipMiddleware`, which, if present, should come first.

Add the following lines at the end of the `settings.py` file:

```
INTERNAL_IPS = [
    '127.0.0.1',
]
```

Django Debug Toolbar will only display if your IP address matches an entry in the `INTERNAL_IPS` setting. To prevent showing debug information in production, Django Debug Toolbar checks that the `DEBUG` setting is `True`.

Edit the main `urls.py` file of your project and add the following URL pattern highlighted in bold to the `urlpatterns`:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
         include('social_django.urls', namespace='social')),
    path('images/', include('images.urls', namespace='images')),
    path('__debug__/', include('debug_toolbar.urls')),
]
```

Django Debug Toolbar is now installed in your project. Let's try it out!

Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/images/` with your browser. You should now see a collapsible sidebar on the right. It should look as follows:

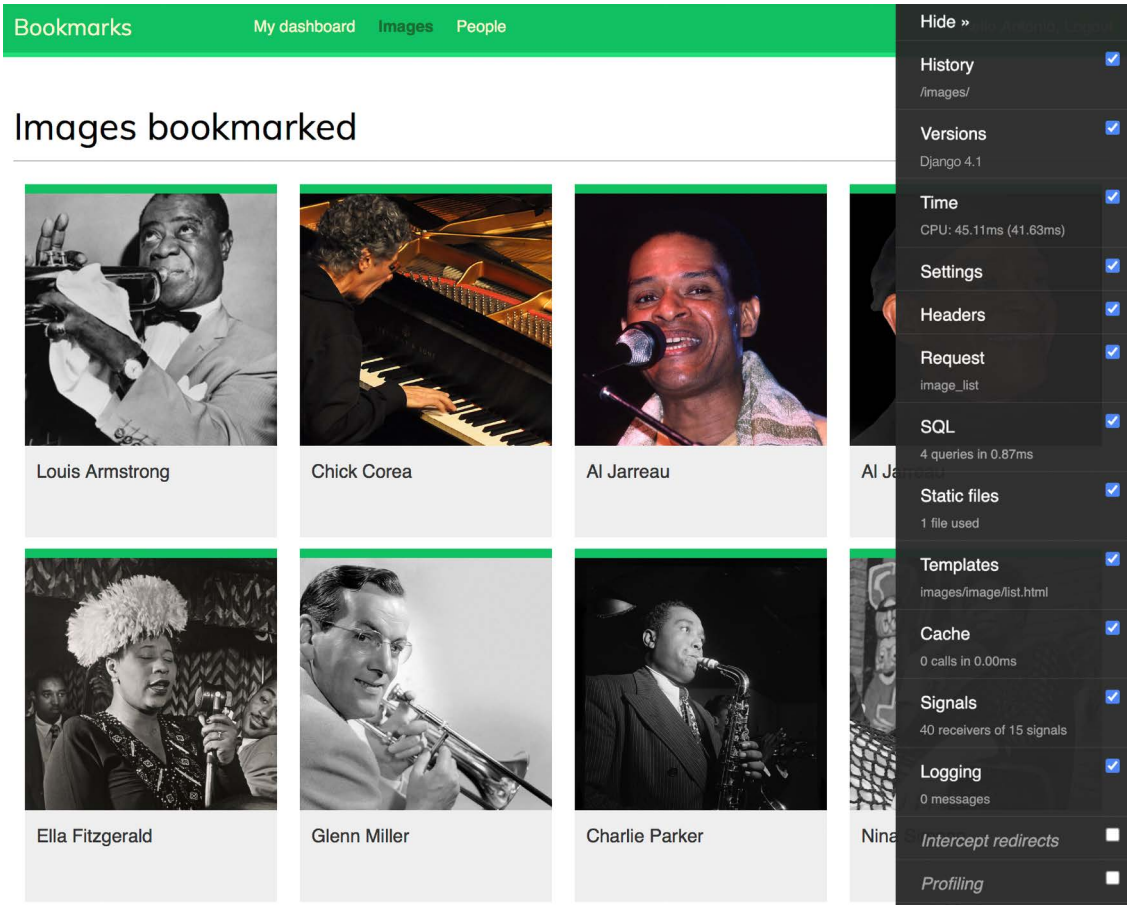


Figure 7.7: The Django Debug Toolbar sidebar

Figure 7.7 image attributions:

- *Chick Corea* by ataelw (license: Creative Commons Attribution 2.0 Generic: `https://creativecommons.org/licenses/by/2.0/`)
- *Al Jarreau – Düsseldorf 1981* by Eddi Laumanns aka RX-Guru (license: Creative Commons Attribution 3.0 Unported: `https://creativecommons.org/licenses/by/3.0/`)
- *Al Jarreau* by Kingkongphoto & www.celebrity-photos.com (license: Creative Commons Attribution-ShareAlike 2.0 Generic: `https://creativecommons.org/licenses/by-sa/2.0/`)

If the debug toolbar doesn't appear, check the RunServer shell console log. If you see a MIME type error, it is most likely that your MIME map files are incorrect or need to be updated.

You can apply the correct mapping for JavaScript and CSS files by adding the following lines to the `settings.py` file:

```
if DEBUG:
    import mimetypes
    mimetypes.add_type('application/javascript', '.js', True)
    mimetypes.add_type('text/css', '.css', True)
```

Django Debug Toolbar panels

Django Debug Toolbar features multiple panels that organize the debug information for the request/response cycle. The sidebar contains links to each panel, and you can use the checkbox of any panel to activate or deactivate it. The change will be applied to the next request. This is useful when we are not interested in a specific panel, but the calculation adds too much overhead to the request.

Click on **Time** in the sidebar menu. You will see the following panel:

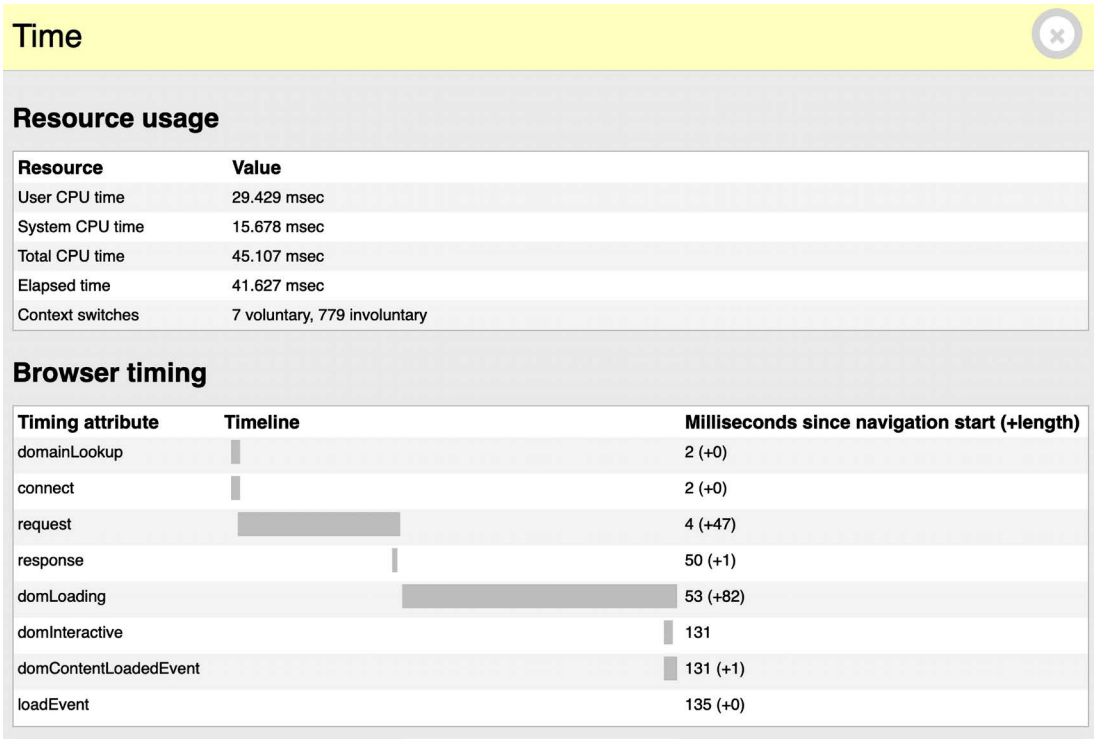


Figure 7.8: Time panel – Django Debug Toolbar

The **Time** panel includes a timer for the different phases of the request/response cycle. It also shows CPU, elapsed time, and the number of context switches. If you are using WIndows, you won't be able to see the **Time** panel. In Windows, only the total time is available and displayed in the toolbar.

Click on **SQL** in the sidebar menu. You will see the following panel:

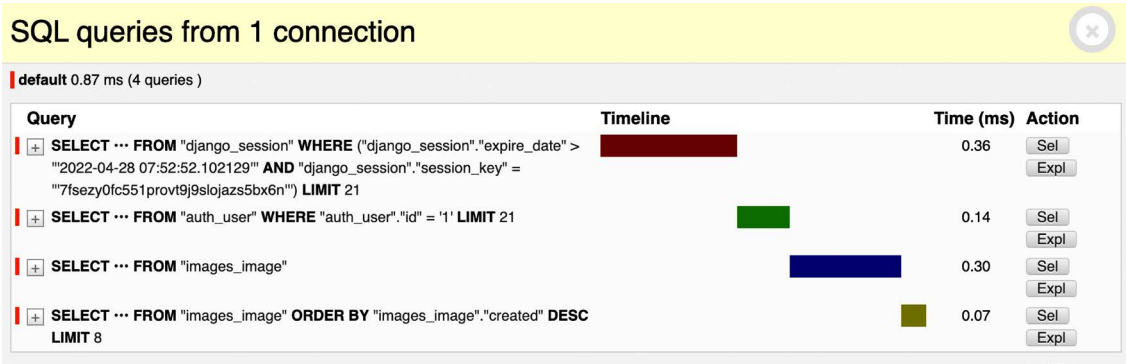


Figure 7.9: SQL panel – Django Debug Toolbar

Here you can see the different SQL queries that have been executed. This information can help you identify unnecessary queries, duplicated queries that can be reused, or long-running queries that can be optimized. Based on your findings, you can improve QuerySets in your views, create new indexes on model fields if necessary, or cache information when needed. In this chapter, you learned how to optimize queries that involve relationships using `select_related()` and `prefetch_related()`. You will learn how to cache data in *Chapter 14*, *Rendering and Caching Content*.

Click on **Templates** in the sidebar menu. You will see the following panel:

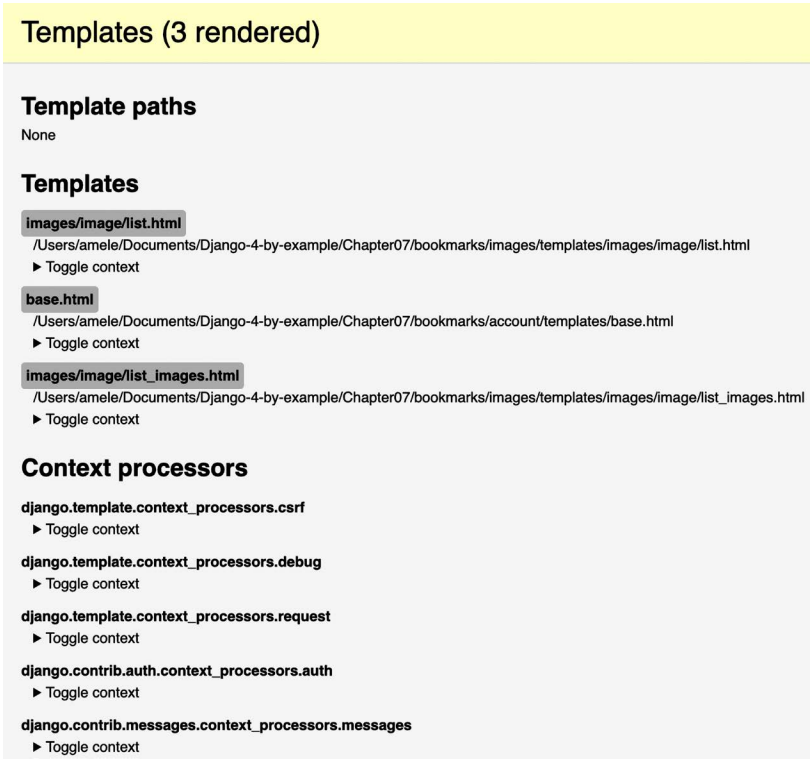


Figure 7.10: Templates panel – Django Debug Toolbar

This panel shows the different templates used when rendering the content, the template paths, and the context used. You can also see the different context processors used. You will learn about context processors in *Chapter 8*, *Building an Online Shop*.

Click on **Signals** in the sidebar menu. You will see the following panel:

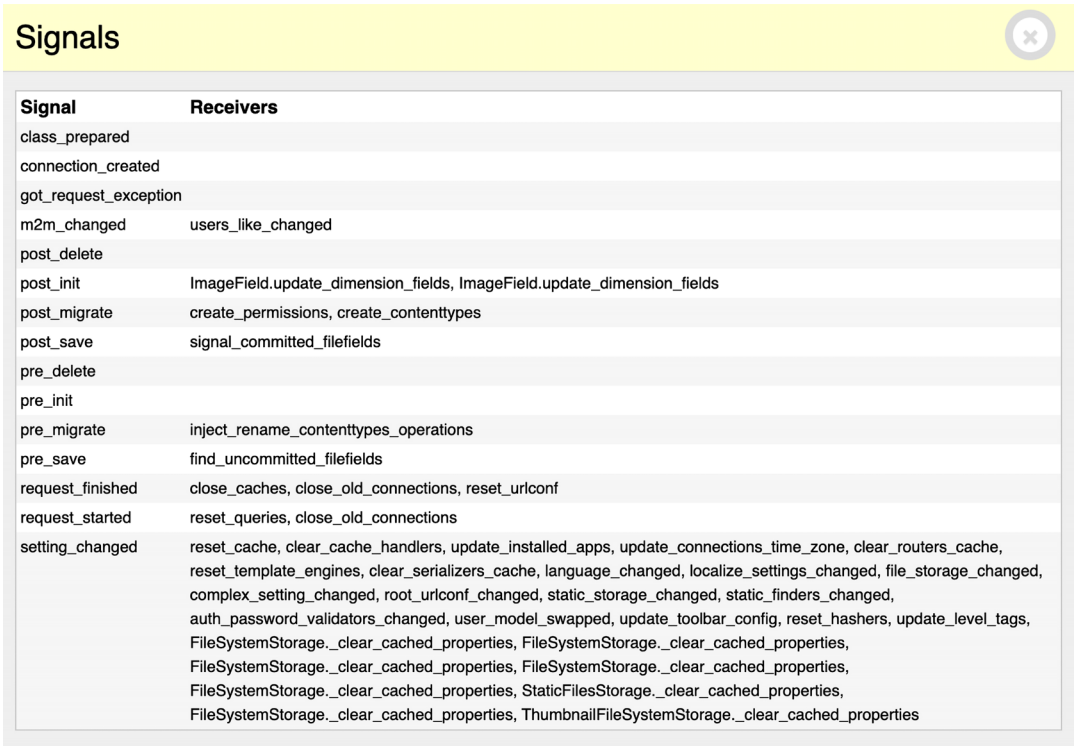


Figure 7.11: Signals panel – Django Debug Toolbar

In this panel, you can see all the signals that are registered in your project and the receiver functions attached to each signal. For example, you can find the `users_like_changed` receiver function you created before, attached to the `m2m_changed` signal. The other signals and receivers are part of the different Django applications.

We have reviewed some of the panels that ship with Django Debug Toolbar. Besides the built-in panels, you can find additional third-party panels that you can download and use at `https://django-debug-toolbar.readthedocs.io/en/latest/panels.html#third-party-panels`.

Django Debug Toolbar commands

Besides the request/response debug panels, Django Debug Toolbar provides a management command to debug SQL for ORM calls. The management command `debugsqlshell` replicates the Django `shell` command but it outputs SQL statements for queries performed with the Django ORM.

Open the shell with the following command:

```
python manage.py debugsqlshell
```

Execute the following code:

```
>>> from images.models import Image
>>> Image.objects.get(id=1)
```

You will see the following output:

```
SELECT "images_image"."id",
       "images_image"."user_id",
       "images_image"."title",
       "images_image"."slug",
       "images_image"."url",
       "images_image"."image",
       "images_image"."description",
       "images_image"."created",
       "images_image"."total_likes"
FROM "images_image"
WHERE "images_image"."id" = 1
LIMIT 21 [0.44ms]
<Image: Django and Duke>
```

You can use this command to test ORM queries before adding them to your views. You can check the resulting SQL statement and the execution time for each ORM call.

In the next section, you will learn how to count image views using Redis, an in-memory database that provides low latency and high-throughput data access.

Counting image views with Redis

Redis is an advanced key/value database that allows you to save different types of data. It also has extremely fast I/O operations. Redis stores everything in memory, but the data can be persisted by dumping the dataset to disk every once in a while, or by adding each command to a log. Redis is very versatile compared to other key/value stores: it provides a set of powerful commands and supports diverse data structures, such as strings, hashes, lists, sets, ordered sets, and even bitmaps or HyperLogLogs.

Although SQL is best suited to schema-defined persistent data storage, Redis offers numerous advantages when dealing with rapidly changing data, volatile storage, or when a quick cache is needed. Let's take a look at how Redis can be used to build new functionality into your project.

You can find more information about Redis on its homepage at `https://redis.io/`.

Redis provides a Docker image that makes it very easy to deploy a Redis server with a standard configuration.

Installing Docker

Docker is a popular open-source containerization platform. It enables developers to package applications into containers, simplifying the process of building, running, managing, and distributing applications.

First, download and install Docker for your OS. You will find instructions for downloading and installing Docker on Linux, macOS, and Windows at `https://docs.docker.com/get-docker/`.

Installing Redis

After installing Docker on your Linux, macOS, or Windows machine, you can easily pull the Redis Docker image. Run the following command from the shell:

```
docker pull redis
```

This will download the Redis Docker image to your local machine. You can find information about the official Redis Docker image at `https://hub.docker.com/_/redis`. You can find other alternative methods to install Redis at `https://redis.io/download/`.

Execute the following command in the shell to start the Redis Docker container:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

With this command, we run Redis in a Docker container. The `-it` option tells Docker to take you straight inside the container for interactive input. The `--rm` option tells Docker to automatically clean up the container and remove the file system when the container exits. The `--name` option is used to assign a name to the container. The `-p` option is used to publish the `6379` port, on which Redis runs, to the same host interface port. `6379` is the default port for Redis.

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

```
# Server initialized
* Ready to accept connections
```

Keep the Redis server running on port `6379` and open another shell. Start the Redis client with the following command:

```
docker exec -it redis sh
```

You will see a line with the hash symbol:

```
#
```

Start the Redis client with the following command:

```
# redis-cli
```

You will see the Redis client shell prompt, like this:

```
127.0.0.1:6379>
```

The Redis client allows you to execute Redis commands directly from the shell. Let's try some commands. Enter the `SET` command in the Redis shell to store a value in a key:

```
127.0.0.1:6379> SET name "Peter"
OK
```

The preceding command creates a `name` key with the string value `"Peter"` in the Redis database. The `OK` output indicates that the key has been saved successfully.

Next, retrieve the value using the `GET` command, as follows:

```
127.0.0.1:6379> GET name
"Peter"
```

You can also check whether a key exists using the `EXISTS` command. This command returns `1` if the given key exists, and `0` otherwise:

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

You can set the time for a key to expire using the `EXPIRE` command, which allows you to set the time-to-live in seconds. Another option is using the `EXPIREAT` command, which expects a Unix timestamp. Key expiration is useful for using Redis as a cache or to store volatile data:

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379> EXPIRE name 2
(integer) 1
```

Wait for more than two seconds and try to get the same key again:

```
127.0.0.1:6379> GET name
(nil)
```

The `(nil)` response is a null response and means that no key has been found. You can also delete any key using the `DEL` command, as follows:

```
127.0.0.1:6379> SET total 1
OK
127.0.0.1:6379> DEL total
(integer) 1
127.0.0.1:6379> GET total
(nil)
```

These are just basic commands for key operations. You can find all Redis commands at `https://redis.io/commands/` and all Redis data types at `https://redis.io/docs/manual/data-types/`.

Using Redis with Python

You will need Python bindings for Redis. Install `redis-py` via `pip` using the following command:

```
pip install redis==4.3.4
```

You can find the `redis-py` documentation at `https://redis-py.readthedocs.io/`.

The `redis-py` package interacts with Redis, providing a Python interface that follows the Redis command syntax. Open the Python shell with the following command:

```
python manage.py shell
```

Execute the following code:

```
>>> import redis
>>> r = redis.Redis(host='localhost', port=6379, db=0)
```

The preceding code creates a connection with the Redis database. In Redis, databases are identified by an integer index instead of a database name. By default, a client is connected to database `0`. The number of available Redis databases is set to `16`, but you can change this in the `redis.conf` configuration file.

Next, set a key using the Python shell:

```
>>> r.set('foo', 'bar')
True
```

The command returns `True`, indicating that the key has been successfully created. Now you can retrieve the key using the `get()` command:

```
>>> r.get('foo')
b'bar'
```

As you will note from the preceding code, the methods of `Redis` follow the Redis command syntax.

Let's integrate Redis into your project. Edit the `settings.py` file of the `bookmarks` project and add the following settings to it:

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0
```

These are the settings for the Redis server and the database that you will use for your project.

Storing image views in Redis

Let's find a way to store the total number of times an image has been viewed. If you implement this using the Django ORM, it will involve a SQL `UPDATE` query every time an image is displayed.

If you use Redis instead, you just need to increment a counter stored in memory, resulting in much better performance and less overhead.

Edit the `views.py` file of the `images` application and add the following code to it after the existing `import` statements:

```
import redis
from django.conf import settings

# connect to redis
r = redis.Redis(host=settings.REDIS_HOST,
                port=settings.REDIS_PORT,
                db=settings.REDIS_DB)
```

With the preceding code, you establish the Redis connection in order to use it in your views. Edit the `views.py` file of the `images` application and modify the `image_detail` view, like this. The new code is highlighted in bold:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # increment total image views by 1
    total_views = r.incr(f'image:{image.id}:views')
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

In this view, you use the `incr` command, which increments the value of a given key by `1`. If the key doesn't exist, the `incr` command creates it. The `incr()` method returns the final value of the key after performing the operation. You store the value in the `total_views` variable and pass it into the template context. You build the Redis key using a notation such as `object-type:id:field` (for example, `image:33:id`).

The convention for naming Redis keys is to use a colon sign as a separator for creating namespaced keys. By doing so, the key names are especially verbose and related keys share part of the same schema in their names.

Edit the `images/image/detail.html` template of the `images` application and add the following code highlighted in bold:

```
...
<div class="image-info">
  <div>
```

```
    <span class="count">
      <span class="total">{{ total_likes }}</span>
      like{{ total_likes|pluralize }}
    </span>
    <span class="count">
      {{ total_views }} view{{ total_views|pluralize }}
    </span>
    <a href="#" data-id="{{ image.id }}" data-action="{% if request.user in
users_like %}un{% endif %}like"
    class="like button">
      {% if request.user not in users_like %}
        Like
      {% else %}
        Unlike
      {% endif %}
    </a>
  </div>
  {{ image.description|linebreaks }}
</div>
...
```

Run the development server with the following command:

```
python manage.py runserver
```

Open an image detail page in your browser and reload it several times. You will see that each time the view is processed, the total views displayed is incremented by 1. Take a look at the following example:

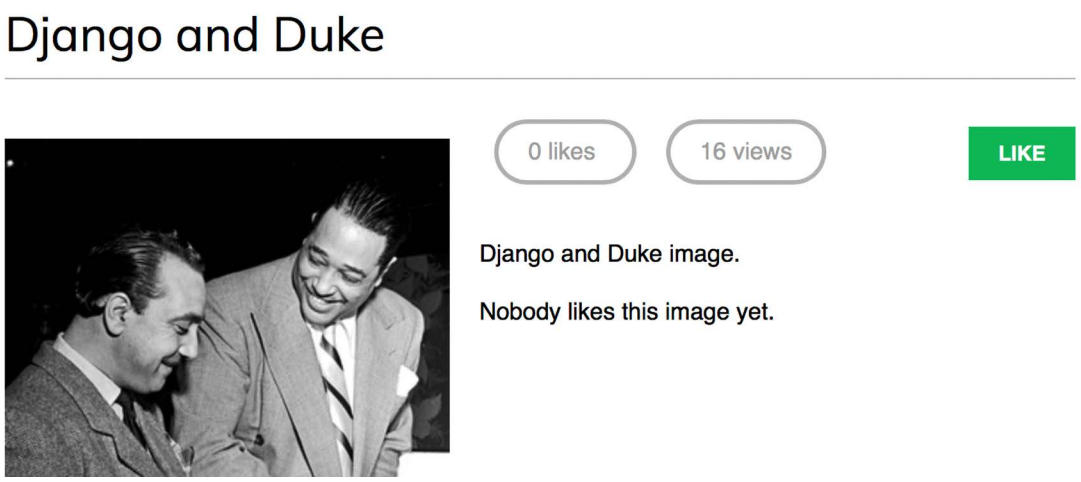


Figure 7.12: The image detail page, including the count of likes and views

Great! You have successfully integrated Redis into your project to count image views. In the next section, you will learn how to build a ranking of the most viewed images with Redis.

Storing a ranking in Redis

We will now create something more complex with Redis. We will use Redis to store a ranking of the most viewed images on the platform. We will use Redis sorted sets for this. A sorted set is a non-repeating collection of strings in which every member is associated with a score. Items are sorted by their score.

Edit the `views.py` file of the `images` application and add the following code highlighted in bold to the `image_detail` view:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # increment total image views by 1
    total_views = r.incr(f'image:{image.id}:views')
    # increment image ranking by 1
    r.zincrby('image_ranking', 1, image.id)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

You use the `zincrby()` command to store image views in a sorted set with the `image:ranking` key. You will store the image `id` and a related score of `1`, which will be added to the total score of this element in the sorted set. This will allow you to keep track of all image views globally and have a sorted set ordered by the total number of views.

Now, create a new view to display the ranking of the most viewed images. Add the following code to the `views.py` file of the `images` application:

```
@login_required
def image_ranking(request):
    # get image ranking dictionary
    image_ranking = r.zrange('image_ranking', 0, -1,
                             desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # get most viewed images
    most_viewed = list(Image.objects.filter(
                           id__in=image_ranking_ids))
    most_viewed.sort(key=lambda x: image_ranking_ids.index(x.id))
    return render(request,
                  'images/image/ranking.html',
                  {'section': 'images',
                   'most_viewed': most_viewed})
```

The `image_ranking` view works like this:

1. You use the `zrange()` command to obtain the elements in the sorted set. This command expects a custom range according to the lowest and highest scores. Using `0` as the lowest and `-1` as the highest score, you are telling Redis to return all elements in the sorted set. You also specify `desc=True` to retrieve the elements ordered by descending score. Finally, you slice the results using `[:10]` to get the first 10 elements with the highest score.
2. You build a list of returned image IDs and store it in the `image_ranking_ids` variable as a list of integers. You retrieve the `Image` objects for those IDs and force the query to be executed using the `list()` function. It is important to force the QuerySet execution because you will use the `sort()` list method on it (at this point, you need a list of objects instead of a QuerySet).
3. You sort the `Image` objects by their index of appearance in the image ranking. Now you can use the `most_viewed` list in your template to display the 10 most viewed images.

Create a new `ranking.html` template inside the `images/image/` template directory of the `images` application and add the following code to it:

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

{% block title %}Images ranking{% endblock %}

{% block content %}
  <h1>Images ranking</h1>
  <ol>
    {% for image in most_viewed %}
      <li>
        <a href="{{ image.get_absolute_url }}">
          {{ image.title }}
        </a>
      </li>
    {% endfor %}
  </ol>
{% endblock %}
```

The template is pretty straightforward. You iterate over the `Image` objects contained in the `most_viewed` list and display their names, including a link to the image detail page.

Finally, you need to create a URL pattern for the new view. Edit the `urls.py` file of the `images` application and add the following URL pattern highlighted in bold:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>/',
         views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
```

```
    path('', views.image_list, name='list'),
    path('ranking/', views.image_ranking, name='ranking'),
]
```

Run the development server, access your site in your web browser, and load the image detail page multiple times for different images. Then, access `http://127.0.0.1:8000/images/ranking/` from your browser. You should be able to see an image ranking, as follows:

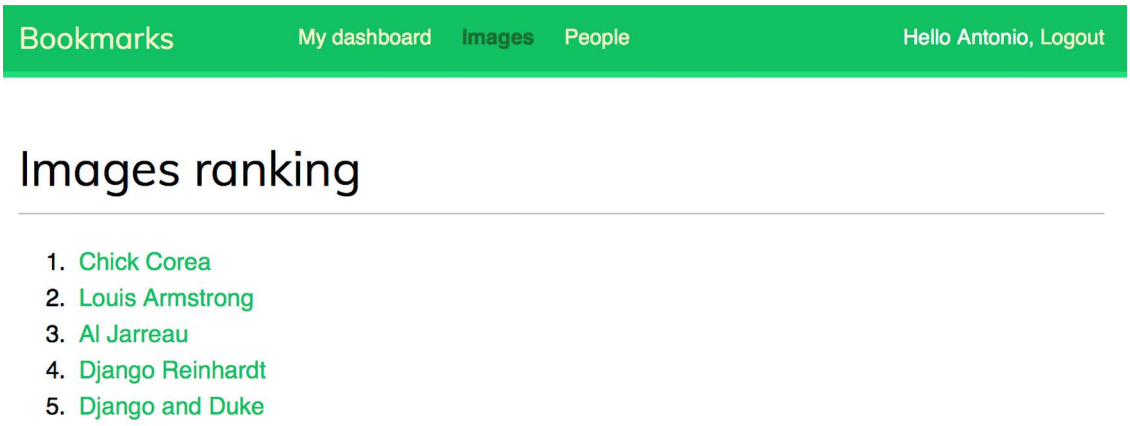


Figure 7.13: The ranking page built with data retrieved from Redis

Great! You just created a ranking with Redis.

Next steps with Redis

Redis is not a replacement for your SQL database, but it does offer fast in-memory storage that is more suitable for certain tasks. Add it to your stack and use it when you really feel it's needed. The following are some scenarios in which Redis could be useful:

- **Counting**: As you have seen, it is very easy to manage counters with Redis. You can use `incr()` and `incrby()` for counting stuff.
- **Storing the latest items**: You can add items to the start/end of a list using `lpush()` and `rpush()`. Remove and return the first/last element using `lpop()`/`rpop()`. You can trim the list's length using `ltrim()` to maintain its length.
- **Queues**: In addition to push and pop commands, Redis offers the blocking of queue commands.
- **Caching**: Using `expire()` and `expireat()` allows you to use Redis as a cache. You can also find third-party Redis cache backends for Django.
- **Pub/sub**: Redis provides commands for subscribing/unsubscribing and sending messages to channels.
- **Rankings and leaderboards**: Redis' sorted sets with scores make it very easy to create leaderboards.
- **Real-time tracking**: Redis's fast I/O makes it perfect for real-time scenarios.

Additional resources

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

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter07`
- Custom user models – `https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#specifying-a-custom-user-model`
- The `contenttypes` framework – `https://docs.djangoproject.com/en/4.1/ref/contrib/contenttypes/`
- Built-in Django signals – `https://docs.djangoproject.com/en/4.1/ref/signals/`
- Application configuration classes – `https://docs.djangoproject.com/en/4.1/ref/applications/`
- Django Debug Toolbar documentation – `https://django-debug-toolbar.readthedocs.io/`
- Django Debug Toolbar third-party panels – `https://django-debug-toolbar.readthedocs.io/en/latest/panels.html#third-party-panels`
- Redis in-memory data store – `https://redis.io/`
- Docker download and install instructions – `https://docs.docker.com/get-docker/`
- Official Redis Docker image — `https://hub.docker.com/_/redis`.
- Redis download options – `https://redis.io/download/`
- Redis commands – `https://redis.io/commands/`
- Redis data types – `https://redis.io/docs/manual/data-types/`
- `redis-py` documentation – `https://redis-py.readthedocs.io/`

Summary

In this chapter, you built a follow system using many-to-many relationships with an intermediary model. You also created an activity stream using generic relations and you optimized QuerySets to retrieve related objects. This chapter then introduced you to Django signals, and you created a signal receiver function to denormalize related object counts. We covered application configuration classes, which you used to load your signal handlers. You added Django Debug Toolbar to your project. You also learned how to install and configure Redis in your Django project. Finally, you used Redis in your project to store item views, and you built an image ranking with Redis.

In the next chapter, you will learn how to build an online shop. You will create a product catalog and build a shopping cart using sessions. You will learn how to create custom context processors. You will also manage customer orders and send asynchronous notifications using Celery and RabbitMQ.

8

Building an Online Shop

In the previous chapter, you created a follow system and built a user activity stream. You also learned how Django signals work and integrated Redis into your project to count image views.

In this chapter, you will start a new Django project that consists of a fully featured online shop. This chapter and the following two chapters will show you how to build the essential functionalities of an e-commerce platform. Your online shop will enable clients to browse products, add them to the cart, apply discount codes, go through the checkout process, pay with a credit card, and obtain an invoice. You will also implement a recommendation engine to recommend products to your customers, and you will use internationalization to offer your site in multiple languages.

In this chapter, you will learn how to:

- Create a product catalog
- Build a shopping cart using Django sessions
- Create custom context processors
- Manage customer orders
- Configure Celery in your project with RabbitMQ as a message broker
- Send asynchronous notifications to customers using Celery
- Monitor Celery using Flower

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter08`.

All Python modules used in this chapter are included in the `requirements.txt` file in the source code that comes along with this chapter. You can follow the instructions to install each Python module below or you can install all requirements at once with the command `pip install -r requirements.txt`.

Creating an online shop project

Let's start with a new Django project to build an online shop. Your users will be able to browse through a product catalog and add products to a shopping cart. Finally, they will be able to check out the cart and place an order. This chapter will cover the following functionalities of an online shop:

- Creating the product catalog models, adding them to the administration site, and building the basic views to display the catalog
- Building a shopping cart system using Django sessions to allow users to keep selected products while they browse the site
- Creating the form and functionality to place orders on the site
- Sending an asynchronous email confirmation to users when they place an order

Open a shell and use the following command to create a new virtual environment for this project within the `env/` directory:

```
python -m venv env/myshop
```

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

```
source env/myshop/bin/activate
```

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

```
.\env\myshop\Scripts\activate
```

The shell prompt will display your active virtual environment, as follows:

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

Install Django in your virtual environment with the following command:

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

Start a new project called `myshop` with an application called `shop` by opening a shell and running the following command:

```
django-admin startproject myshop
```

The initial project structure has been created. Use the following commands to get into your project directory and create a new application named `shop`:

```
cd myshop/
django-admin startapp shop
```

Edit `settings.py` and add the following line highlighted in bold to the `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'shop.apps.ShopConfig',
]
```

Your application is now active for this project. Let's define the models for the product catalog.

Creating product catalog models

The catalog of your shop will consist of products that are organized into different categories. Each product will have a name, an optional description, an optional image, a price, and its availability.

Edit the `models.py` file of the `shop` application that you just created and add the following code:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200,
                            unique=True)

    class Meta:
        ordering = ['name']
        indexes = [
            models.Index(fields=['name']),
        ]
        verbose_name = 'category'
        verbose_name_plural = 'categories'

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category,
                                 related_name='products',
                                 on_delete=models.CASCADE)
    name = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200)
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                              blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10,
                                decimal_places=2)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
```

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

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

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

These are the `Category` and `Product` models. The `Category` model consists of a `name` field and a unique `slug` field (`unique` implies the creation of an index). In the `Meta` class of the `Category` model, we have defined an index for the `name` field.

The `Product` model fields are as follows:

- `category`: A `ForeignKey` to the `Category` model. This is a one-to-many relationship: a product belongs to one category and a category contains multiple products.
- `name`: The name of the product.
- `slug`: The slug for this product to build beautiful URLs.
- `image`: An optional product image.
- `description`: An optional description of the product.
- `price`: This field uses Python's `decimal.Decimal` type to store a fixed-precision decimal number. The maximum number of digits (including the decimal places) is set using the `max_digits` attribute and decimal places with the `decimal_places` attribute.
- `available`: A Boolean value that indicates whether the product is available or not. It will be used to enable/disable the product in the catalog.
- `created`: This field stores when the object was created.
- `updated`: This field stores when the object was last updated.

For the `price` field, we use `DecimalField` instead of `FloatField` to avoid rounding issues.

Always use `DecimalField` to store monetary amounts. `FloatField` uses Python's `float` type internally, whereas `DecimalField` uses Python's `Decimal` type. By using the `Decimal` type, you will avoid `float` rounding issues.

In the `Meta` class of the `Product` model, we have defined a multiple-field index for the `id` and `slug` fields. Both fields are indexed together to improve performance for queries that utilize the two fields.

We plan to query products by both `id` and `slug`. We have added an index for the `name` field and an index for the `created` field. We have used a hyphen before the field name to define the index with a descending order.

Figure 8.1 shows the two data models you have created:

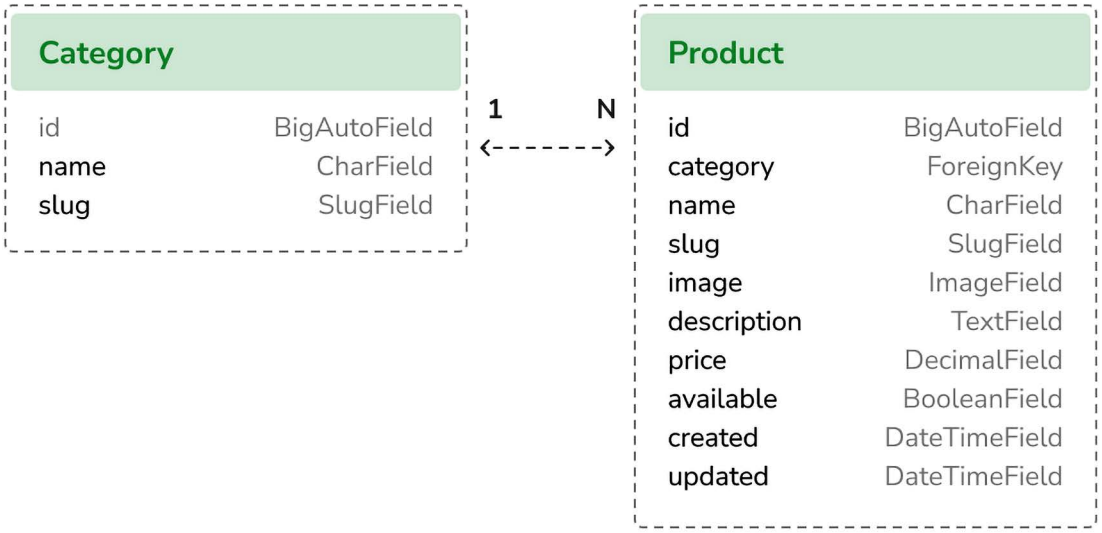


Figure 8.1: Models for the product catalog

In *Figure 8.1*, you can see the different fields of the data models and the one-to-many relationship between the `Category` and the `Product` models.

These models will result in the following database tables displayed in *Figure 8.2*:

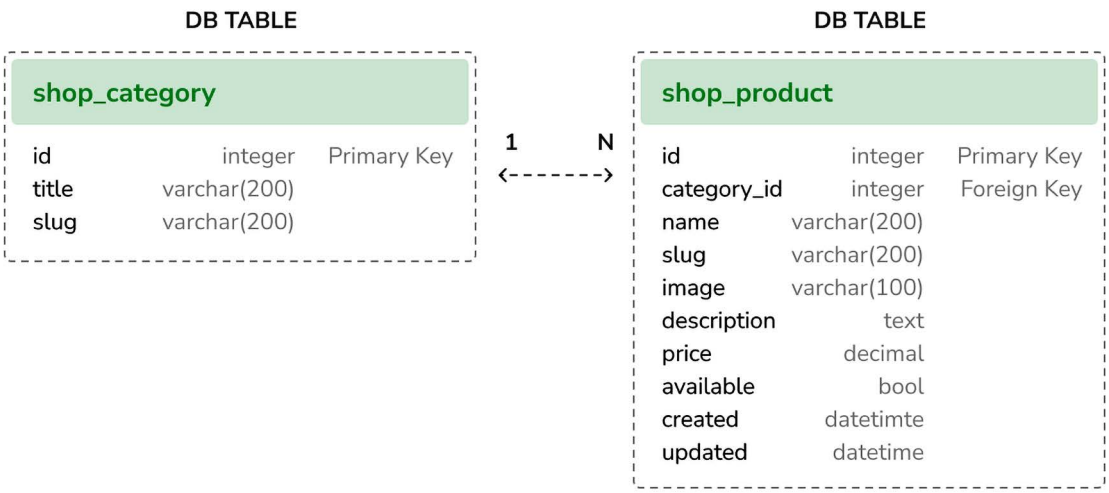


Figure 8.2: Database tables for the product catalog models

The one-to-many relationship between both tables is defined with the `category_id` field in the `shop_product` table, which is used to store the ID of the related `Category` for each `Product` object.

Let's create the initial database migrations for the `shop` application. Since you are going to deal with images in your models you will need to install the Pillow library. Remember that in *Chapter 4*, *Building a Social Website*, you learned how to install the Pillow library to manage images. Open the shell and install `Pillow` with the following command:

```
pip install Pillow==9.2.0
```

Now run the next command to create initial migrations for your project:

```
python manage.py makemigrations
```

You will see the following output:

```
Migrations for 'shop':
  shop/migrations/0001_initial.py
    - Create model Category
    - Create model Product
    - Create index shop_catego_name_289c7e_idx on field(s) name of model
category
    - Create index shop_produc_id_f21274_idx on field(s) id, slug of model
product
    - Create index shop_produc_name_a2070e_idx on field(s) name of model
product
    - Create index shop_produc_created_ef211c_idx on field(s) -created of model
product
```

Run the next command to sync the database:

```
python manage.py migrate
```

You will see output that includes the following line:

```
Applying shop.0001_initial... OK
```

The database is now synced with your models.

Registering catalog models on the administration site

Let's add your models to the administration site so that you can easily manage categories and products. Edit the `admin.py` file of the `shop` application and add the following code to it:

```
from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price',
                    'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
```

Remember that you use the `prepopulated_fields` attribute to specify fields where the value is automatically set using the value of other fields. As you have seen before, this is convenient for generating slugs.

You use the `list_editable` attribute in the `ProductAdmin` class to set the fields that can be edited from the list display page of the administration site. This will allow you to edit multiple rows at once. Any field in `list_editable` must also be listed in the `list_display` attribute, since only the fields displayed can be edited.

Now create a superuser for your site using the following command:

```
python manage.py createsuperuser
```

Enter the desired username, email, and password. Run the development server with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/shop/product/add/` in your browser and log in with the user that you just created. Add a new category and product using the administration interface. The add product form should look as follows:

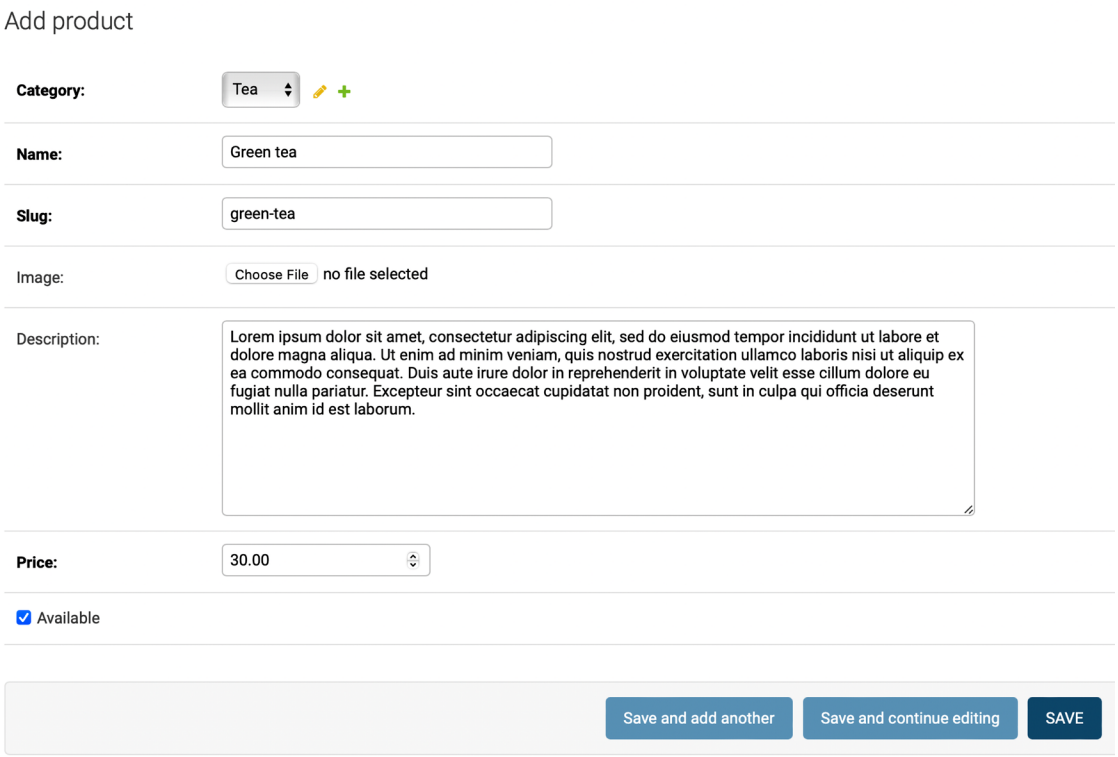


Figure 8.3: The product creation form

Click on the **Save** button. The product change list page of the administration page will then look like this:

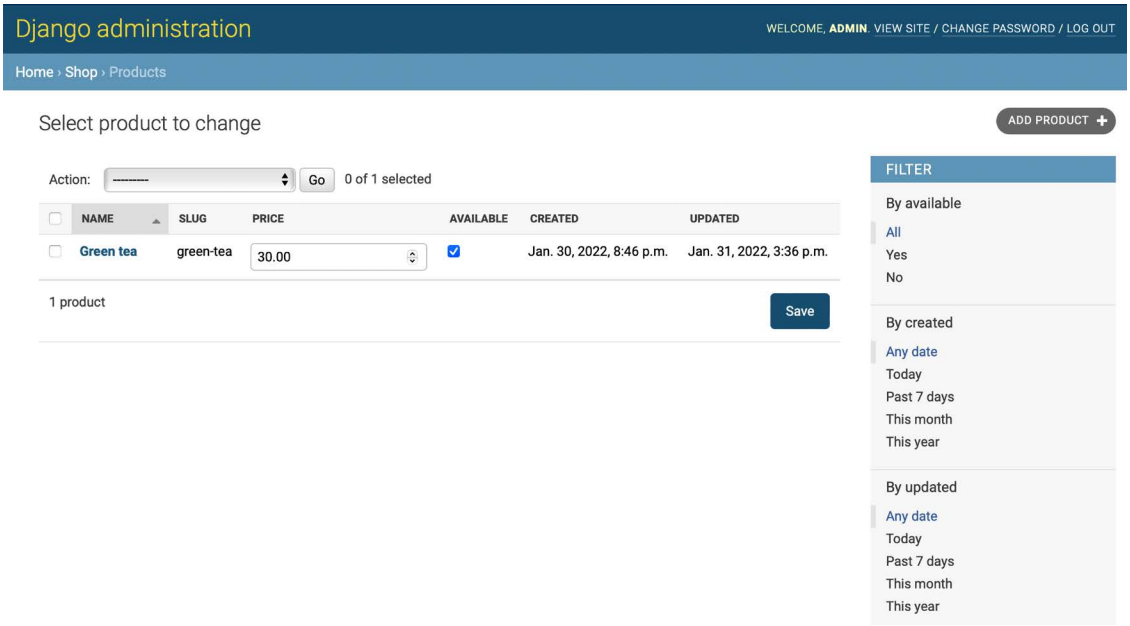


Figure 8.4: The product change list page

Building catalog views

In order to display the product catalog, you need to create a view to list all the products or filter products by a given category. Edit the `views.py` file of the `shop` application and add the following code highlighted in bold:

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
```

```
        category = get_object_or_404(Category,
                                     slug=category_slug)
        products = products.filter(category=category)
    return render(request,
                  'shop/product/list.html',
                  {'category': category,
                   'categories': categories,
                   'products': products})
```

In the preceding code, you filter the QuerySet with `available=True` to retrieve only available products. You use an optional `category_slug` parameter to optionally filter products by a given category.

You also need a view to retrieve and display a single product. Add the following view to the `views.py` file:

```
def product_detail(request, id, slug):
    product = get_object_or_404(Product,
                                id=id,
                                slug=slug,
                                available=True)
    return render(request,
                  'shop/product/detail.html',
                  {'product': product})
```

The `product_detail` view expects the `id` and `slug` parameters in order to retrieve the `Product` instance. You can get this instance just through the ID, since it's a unique attribute. However, you include the slug in the URL to build SEO-friendly URLs for products.

After building the product list and detail views, you have to define URL patterns for them. Create a new file inside the `shop` application directory and name it `urls.py`. Add the following code to it:

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

app_name = 'shop'

urlpatterns = [
    path('', views.product_list, name='product_list'),
    path('<slug:category_slug>/', views.product_list,
         name='product_list_by_category'),
    path('<int:id>/<slug:slug>/', views.product_detail,
         name='product_detail'),
]
```

These are the URL patterns for your product catalog. You have defined two different URL patterns for the `product_list` view: a pattern named `product_list`, which calls the `product_list` view without any parameters, and a pattern named `product_list_by_category`, which provides a `category_slug` parameter to the view for filtering products according to a given category. You added a pattern for the `product_detail` view, which passes the `id` and `slug` parameters to the view in order to retrieve a specific product.

Edit the `urls.py` file of the `myshop` project to make it look like this:

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

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

In the main URL patterns of the project, you include URLs for the `shop` application under a custom namespace named `shop`.

Next, edit the `models.py` file of the `shop` application, import the `reverse()` function, and add a `get_absolute_url()` method to the `Category` and `Product` models as follows. The new code is highlighted in bold:

```
from django.db import models
from django.urls import reverse

class Category(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_list_by_category',
                       args=[self.slug])

class Product(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_detail',
                       args=[self.id, self.slug])
```

As you already know, `get_absolute_url()` is the convention to retrieve the URL for a given object. Here, you use the URL patterns that you just defined in the `urls.py` file.

Creating catalog templates

Now you need to create templates for the product list and detail views. Create the following directory and file structure inside the `shop` application directory:

```
templates/
    shop/
        base.html
        product/
            list.html
            detail.html
```

You need to define a base template and then extend it in the product list and detail templates. Edit the `shop/base.html` template and add the following code to it:

```
{% load static %}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>{% block title %}My shop{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <a href="/" class="logo">My shop</a>
    </div>
    <div id="subheader">
      <div class="cart">
        Your cart is empty.
      </div>
    </div>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
  </body>
</html>
```

This is the base template that you will use for your shop. In order to include the CSS styles and images that are used by the templates, you need to copy the static files that accompany this chapter, which are located in the `static/` directory of the `shop` application. Copy them to the same location in your project. You can find the contents of the directory at `https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter08/myshop/shop/static`.

Edit the `shop/product/list.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
  {% if category %}{{ category.name }}{% else %}Products{% endif %}
{% endblock %}

{% block content %}
  <div id="sidebar">
    <h3>Categories</h3>
    <ul>
      <li {% if not category %}class="selected"{% endif %}>
        <a href="{% url "shop:product_list" %}">All</a>
      </li>
      {% for c in categories %}
        <li {% if category.slug == c.slug %}class="selected"
        {% endif %}>
          <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
  <div id="main" class="product-list">
    <h1>{% if category %}{{ category.name }}{% else %}Products
    {% endif %}</h1>
    {% for product in products %}
      <div class="item">
        <a href="{{ product.get_absolute_url }}">
          <img src="{% if product.image %}{{ product.image.url }}{% else %}{%
static "img/no_image.png" %}{% endif %}">
        </a>
        <a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
        <br>
        ${{ product.price }}
      </div>
    {% endfor %}
  </div>
{% endblock %}
```

Make sure that no template tag is split into multiple lines.

This is the product list template. It extends the `shop/base.html` template and uses the `categories` context variable to display all the categories in a sidebar, and `products` to display the products of the current page. The same template is used for both listing all available products and listing products filtered by a category. Since the `image` field of the `Product` model can be blank, you need to provide a default image for the products that don't have an image. The image is located in your static files directory with the relative path `img/no_image.png`.

Since you are using `ImageField` to store product images, you need the development server to serve uploaded image files.

Edit the `settings.py` file of `myshop` and add the following settings:

```
MEDIA_URL = 'media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

`MEDIA_URL` is the base URL that serves media files uploaded by users. `MEDIA_ROOT` is the local path where these files reside, which you build by dynamically prepending the `BASE_DIR` variable.

For Django to serve the uploaded media files using the development server, edit the main `urls.py` file of `myshop` and add the following code highlighted in bold:

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

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

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Remember that you only serve static files this way during development. In a production environment, you should never serve static files with Django; the Django development server doesn't serve static files in an efficient manner. *Chapter 17*, *Going Live*, will teach you how to serve static files in a production environment.

Run the development server with the following command:

```
python manage.py runserver
```

Add a couple of products to your shop using the administration site and open `http://127.0.0.1:8000/` in your browser. You will see the product list page, which will look similar to this:

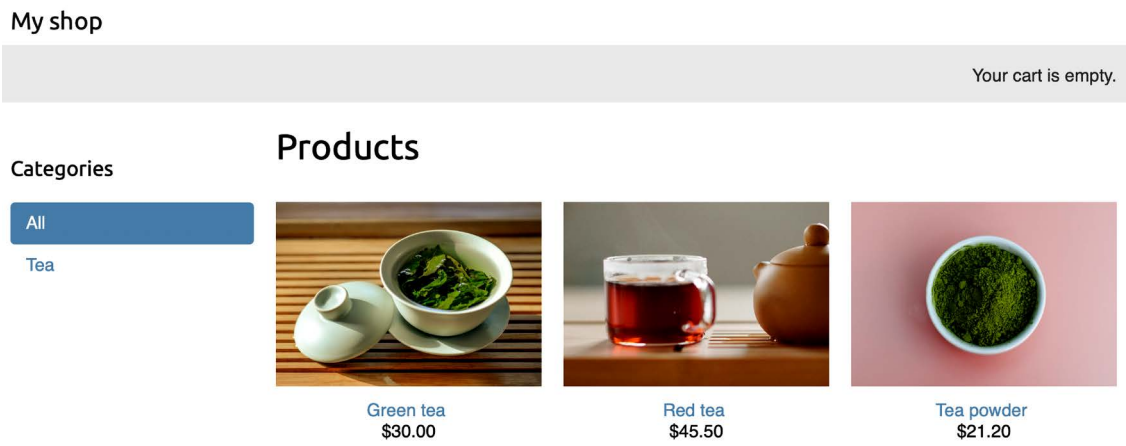


Figure 8.5: The product list page

Images in this chapter:

- *Green tea*: Photo by Jia Ye on Unsplash
- *Red tea*: Photo by Manki Kim on Unsplash
- *Tea powder*: Photo by Phuong Nguyen on Unsplash

If you create a product using the administration site and don't upload any image for it, the default `no_image.png` image will be displayed instead:

Figure 8.6: The product list displaying a default image for products that have no image

Edit the `shop/product/detail.html` template and add the following code to it:

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
  {{ product.name }}
{% endblock %}
{% block content %}
  <div class="product-detail">
    <img src="{% if product.image %}{{ product.image.url }}{% else %}
    {% static "img/no_image.png" %}{% endif %}">
    <h1>{{ product.name }}</h1>
    <h2>
      <a href="{{ product.category.get_absolute_url }}">
        {{ product.category }}
      </a>
    </h2>
    <p class="price">${{ product.price }}</p>
    {{ product.description|linebreaks }}
  </div>
{% endblock %}
```

In the preceding code, you call the `get_absolute_url()` method on the related category object to display the available products that belong to the same category.

Now open `http://127.0.0.1:8000/` in your browser and click on any product to see the product detail page. It will look as follows:

Figure 8.7: The product detail page

You have now created a basic product catalog. Next, you will implement a shopping cart that allows users to add any product to it while browsing the online shop.

Building a shopping cart

After building the product catalog, the next step is to create a shopping cart so that users can pick the products that they want to purchase. A shopping cart allows users to select products and set the amount they want to order, and then store this information temporarily while they browse the site, until they eventually place an order. The cart has to be persisted in the session so that the cart items are maintained during a user's visit.

You will use Django's session framework to persist the cart. The cart will be kept in the session until it finishes or the user checks out of the cart. You will also need to build additional Django models for the cart and its items.

Using Django sessions

Django provides a session framework that supports anonymous and user sessions. The session framework allows you to store arbitrary data for each visitor. Session data is stored on the server side, and cookies contain the session ID unless you use the cookie-based session engine. The session middleware manages the sending and receiving of cookies. The default session engine stores session data in the database, but you can choose other session engines.

To use sessions, you have to make sure that the `MIDDLEWARE` setting of your project contains `'django.contrib.sessions.middleware.SessionMiddleware'`. This middleware manages sessions. It's added by default to the `MIDDLEWARE` setting when you create a new project using the `startproject` command.

The session middleware makes the current session available in the `request` object. You can access the current session using `request.session`, treating it like a Python dictionary to store and retrieve session data. The `session` dictionary accepts any Python object by default that can be serialized to JSON. You can set a variable in the session like this:

```
request.session['foo'] = 'bar'
```

Retrieve a session key as follows:

```
request.session.get('foo')
```

Delete a key you previously stored in the session as follows:

```
del request.session['foo']
```

When users log in to the site, their anonymous session is lost, and a new session is created for authenticated users. If you store items in an anonymous session that you need to keep after the user logs in, you will have to copy the old session data into the new session. You can do this by retrieving the session data before you log in the user using the `login()` function of the Django authentication system and storing it in the session after that.

Session settings

There are several settings you can use to configure sessions for your project. The most important is `SESSION_ENGINE`. This setting allows you to set the place where sessions are stored. By default, Django stores sessions in the database using the `Session` model of the `django.contrib.sessions` application.

Django offers the following options for storing session data:

- **Database sessions**: Session data is stored in the database. This is the default session engine.
- **File-based sessions**: Session data is stored in the filesystem.
- **Cached sessions**: Session data is stored in a cache backend. You can specify cache backends using the `CACHES` setting. Storing session data in a cache system provides the best performance.
- **Cached database sessions**: Session data is stored in a write-through cache and database. Reads only use the database if the data is not already in the cache.
- **Cookie-based sessions**: Session data is stored in the cookies that are sent to the browser.

For better performance use a cache-based session engine. Django supports Memcached out of the box and you can find third-party cache backends for Redis and other cache systems.

You can customize sessions with specific settings. Here are some of the important session-related settings:

- `SESSION_COOKIE_AGE`: The duration of session cookies in seconds. The default value is `1209600` (two weeks).
- `SESSION_COOKIE_DOMAIN`: The domain used for session cookies. Set this to `mydomain.com` to enable cross-domain cookies or use `None` for a standard domain cookie.
- `SESSION_COOKIE_HTTPONLY`: Whether to use `HttpOnly` flag on the session cookie. If this is set to `True`, client-side JavaScript will not be able to access the session cookie. The default value is `True` for increased security against user session hijacking.
- `SESSION_COOKIE_SECURE`: A Boolean indicating that the cookie should only be sent if the connection is an HTTPS connection. The default value is `False`.
- `SESSION_EXPIRE_AT_BROWSER_CLOSE`: A Boolean indicating that the session has to expire when the browser is closed. The default value is `False`.
- `SESSION_SAVE_EVERY_REQUEST`: A Boolean that, if `True`, will save the session to the database on every request. The session expiration is also updated each time it's saved. The default value is `False`.

You can see all the session settings and their default values at `https://docs.djangoproject.com/en/4.1/ref/settings/#sessions`.

Session expiration

You can choose to use browser-length sessions or persistent sessions using the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting. This is set to `False` by default, forcing the session duration to the value stored in the `SESSION_COOKIE_AGE` setting. If you set `SESSION_EXPIRE_AT_BROWSER_CLOSE` to `True`, the session will expire when the user closes the browser, and the `SESSION_COOKIE_AGE` setting will not have any effect.

You can use the `set_expiry()` method of `request.session` to overwrite the duration of the current session.

Storing shopping carts in sessions

You need to create a simple structure that can be serialized to JSON for storing cart items in a session. The cart has to include the following data for each item contained in it:

- The ID of a `Product` instance
- The quantity selected for the product
- The unit price for the product

Since product prices may vary, let's take the approach of storing the product's price along with the product itself when it's added to the cart. By doing so, you use the current price of the product when users add it to their cart, no matter whether the product's price is changed afterward. This means that the price that the item has when the client adds it to the cart is maintained for that client in the session until checkout is completed or the session finishes.

Next, you have to build functionality to create shopping carts and associate them with sessions. This has to work as follows:

- When a cart is needed, you check whether a custom session key is set. If no cart is set in the session, you create a new cart and save it in the cart session key.
- For successive requests, you perform the same check and get the cart items from the cart session key. You retrieve the cart items from the session and their related `Product` objects from the database.

Edit the `settings.py` file of your project and add the following setting to it:

```
CART_SESSION_ID = 'cart'
```

This is the key that you are going to use to store the cart in the user session. Since Django sessions are managed per visitor, you can use the same cart session key for all sessions.

Let's create an application for managing shopping carts. Open the terminal and create a new application, running the following command from the project directory:

```
python manage.py startapp cart
```

Then, edit the `settings.py` file of your project and add the new application to the `INSTALLED_APPS` setting with the following line highlighted in bold:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
    'cart.apps.CartConfig',
]
```

Create a new file inside the `cart` application directory and name it `cart.py`. Add the following code to it:

```
from decimal import Decimal
from django.conf import settings
from shop.models import Product

class Cart:
    def __init__(self, request):
        """
        Initialize the cart.
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # save an empty cart in the session
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart
```

This is the `Cart` class that will allow you to manage the shopping cart. You require the cart to be initialized with a `request` object. You store the current session using `self.session = request.session` to make it accessible to the other methods of the `Cart` class.

First, you try to get the cart from the current session using `self.session.get(settings.CART_SESSION_ID)`. If no cart is present in the session, you create an empty cart by setting an empty dictionary in the session.

You will build your `cart` dictionary with product IDs as keys, and for each product key, a dictionary will be a value that includes quantity and price. By doing this, you can guarantee that a product will not be added more than once to the cart. This way, you can also simplify retrieving cart items.

Let's create a method to add products to the cart or update their quantity. Add the following `add()` and `save()` methods to the `Cart` class:

```
class Cart:
    # ...
    def add(self, product, quantity=1, override_quantity=False):
```

```
            """
            Add a product to the cart or update its quantity.
            """
            product_id = str(product.id)
            if product_id not in self.cart:
                self.cart[product_id] = {'quantity': 0,
                                         'price': str(product.price)}
            if override_quantity:
                self.cart[product_id]['quantity'] = quantity
            else:
                self.cart[product_id]['quantity'] += quantity
            self.save()

        def save(self):
            # mark the session as "modified" to make sure it gets saved
            self.session.modified = True
```

The `add()` method takes the following parameters as input:

- `product`: The `product` instance to add or update in the cart.
- `quantity`: An optional integer with the product quantity. This defaults to `1`.
- `override_quantity`: This is a Boolean that indicates whether the quantity needs to be overridden with the given quantity (`True`), or whether the new quantity has to be added to the existing quantity (`False`).

You use the product ID as a key in the cart's content dictionary. You convert the product ID into a string because Django uses JSON to serialize session data, and JSON only allows string key names. The product ID is the key, and the value that you persist is a dictionary with quantity and price figures for the product. The product's price is converted from decimal into a string to serialize it. Finally, you call the `save()` method to save the cart in the session.

The `save()` method marks the session as modified using `session.modified = True`. This tells Django that the session has changed and needs to be saved.

You also need a method for removing products from the cart. Add the following method to the `Cart` class:

```
class Cart:
    # ...
    def remove(self, product):
        """
        Remove a product from the cart.
        """
        product_id = str(product.id)
```

```
        if product_id in self.cart:
            del self.cart[product_id]
            self.save()
```

The `remove()` method removes a given product from the `cart` dictionary and calls the `save()` method to update the cart in the session.

You will have to iterate through the items contained in the cart and access the related `Product` instances. To do so, you can define an `__iter__()` method in your class. Add the following method to the `Cart` class:

```
class Cart:
    # ...
    def __iter__(self):
        """
        Iterate over the items in the cart and get the products
        from the database.
        """
        product_ids = self.cart.keys()
        # get the product objects and add them to the cart
        products = Product.objects.filter(id__in=product_ids)
        cart = self.cart.copy()
        for product in products:
            cart[str(product.id)]['product'] = product
        for item in cart.values():
            item['price'] = Decimal(item['price'])
            item['total_price'] = item['price'] * item['quantity']
            yield item
```

In the `__iter__()` method, you retrieve the `Product` instances that are present in the cart to include them in the cart items. You copy the current cart in the `cart` variable and add the `Product` instances to it. Finally, you iterate over the cart items, converting each item's price back into decimal, and adding a `total_price` attribute to each item. This `__iter__()` method will allow you to easily iterate over the items in the cart in views and templates.

You also need a way to return the number of total items in the cart. When the `len()` function is executed on an object, Python calls its `__len__()` method to retrieve its length. Next, you are going to define a custom `__len__()` method to return the total number of items stored in the cart.

Add the following `__len__()` method to the `Cart` class:

```
class Cart:
    # ...
    def __len__(self):
        """
```

```
        Count all items in the cart.
        """
        return sum(item['quantity'] for item in self.cart.values())
```

You return the sum of the quantities of all the cart items.

Add the following method to calculate the total cost of the items in the cart:

```
class Cart:
    # ...
    def get_total_price(self):
        return sum(Decimal(item['price']) * item['quantity'] for item in self.
cart.values())
```

Finally, add a method to clear the cart session:

```
class Cart:
    # ...
    def clear(self):
        # remove cart from session
        del self.session[settings.CART_SESSION_ID]
        self.save()
```

Your `Cart` class is now ready to manage shopping carts.

Creating shopping cart views

Now that you have a `Cart` class to manage the cart, you need to create the views to add, update, or remove items from it. You need to create the following views:

- A view to add or update items in the cart that can handle current and new quantities
- A view to remove items from the cart
- A view to display cart items and totals

Adding items to the cart

To add items to the cart, you need a form that allows the user to select a quantity. Create a `forms.py` file inside the `cart` application directory and add the following code to it:

```
from django import forms

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
                                choices=PRODUCT_QUANTITY_CHOICES,
                                coerce=int)
```

```
    override = forms.BooleanField(required=False,
                                  initial=False,
                                  widget=forms.HiddenInput)
```

You will use this form to add products to the cart. Your `CartAddProductForm` class contains the following two fields:

- `quantity`: This allows the user to select a quantity between 1 and 20. You use a `TypedChoiceField` field with `coerce=int` to convert the input into an integer.
- `override`: This allows you to indicate whether the quantity has to be added to any existing quantity in the cart for this product (`False`), or whether the existing quantity has to be overridden with the given quantity (`True`). You use a `HiddenInput` widget for this field, since you don't want to display it to the user.

Let's create a view for adding items to the cart. Edit the `views.py` file of the `cart` application and add the following code highlighted in bold:

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from shop.models import Product
from .cart import Cart
from .forms import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product,
                 quantity=cd['quantity'],
                 override_quantity=cd['override'])
    return redirect('cart:cart_detail')
```

This is the view for adding products to the cart or updating quantities for existing products. You use the `require_POST` decorator to allow only `POST` requests. The view receives the product ID as a parameter. You retrieve the `Product` instance with the given ID and validate `CartAddProductForm`. If the form is valid, you either add or update the product in the cart. The view redirects to the `cart_detail` URL, which will display the contents of the cart. You are going to create the `cart_detail` view shortly.

You also need a view to remove items from the cart. Add the following code to the `views.py` file of the `cart` application:

```
@require_POST
def cart_remove(request, product_id):
```