Activity Solutions

Chapter 1, An Introduction to Django

Activity 1.01 – creating a Site Welcome Screen

The following steps will help you complete this activity:

1. The `index` view in `views.py` should now just be one line, which returns the rendered template:

```
def index(request):
    return render(request, "base.html")
```

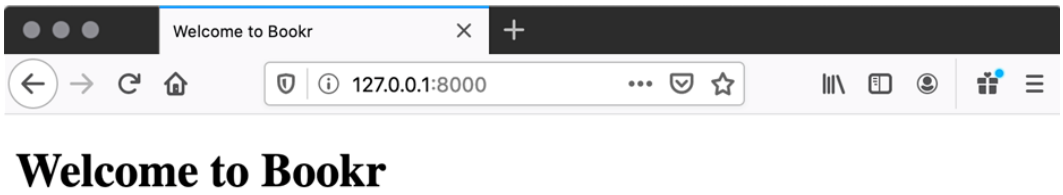
2. Update `base.html`; the `<title>` tag should contain `Welcome to Bookr`:

```
<head>
    <meta charset="UTF-8">
    <title>Welcome to Bookr</title>
</head>
```

The `<body>` tag should contain an `h1` tag with the same message:

```
<body>
<h1>Welcome to Bookr</h1>
</body>
```

The index page of your site should now look like *Activity Figure 1.1*:



Activity Figure 1.1: The Bookr welcome screen

You have created a welcome splash page for Bookr, to which we will be able to add links to other parts of the site as we build them.

Activity 1.02 – book search scaffold

The following steps will help you complete this activity:

1. Create a new file HTML file in the `templates` directory named `search-results.html`. Use a variable (`search_text`) in the `<title>` tag:

```
<head>
    <meta charset="UTF-8">
    <title>Search Results: {{ search_text }}</title>
</head>
```

   Use the same `search_text` variable in an `<h1>` tag inside the body:

```
<body>
<h1>Search Results for <em>{{ search_text }}</em></h1>
</body>
```

2. Create a new function in `views.py` called `book_search`. It should create a variable called `search_text`, which is set from the `search` parameter in the URL:

```
def book_search(request):
    search_text = request.GET.get("search", "")
```

   If no `search` parameter is provided, it will default to an empty string.

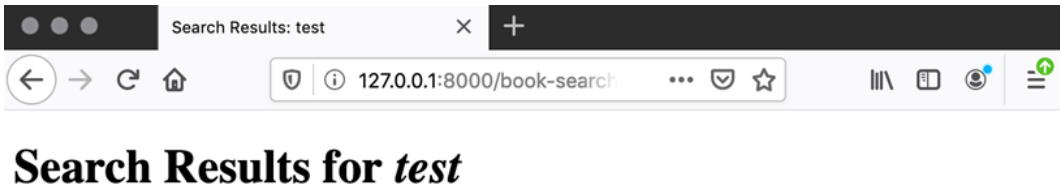
3. Then, render the `search-results.html` template with `search_text` in the context:

```
    return render(request, "search-results.html",
    {"search_text": search_text})
```

4. Add a URL mapping to `urlpatterns` in `urls.py`:

```
    path('book-search', reviews.views.book_search)
```

   Visit `http://127.0.0.1:8000/book-search?search=test` to see the output for the `test` search string (*Activity Figure 1.2*):



Activity Figure 1.2: Searching for the test string

Try changing `test` to other values to see it in action – for example, `Web Development with Django`. Also, try the `</html>` text to see how HTML entities are automatically escaped when rendered in a template. Refer to *Activity Figure 1.1* and *Activity Figure 1.2* to see how your page should look for the latter two examples.

Chapter 2, Models and Migrations

Activity 2.01 – creating models for a project management application

First, create a separate project in PyCharm, and then follow these instructions to complete the activity:

1. Create the `juggler` project by running the following command in the shell:

```
django-admin startproject juggler
```

2. Create an app called `projectm`:

```
django-admin startapp projectm
```

3. Add the app projects in `juggler/settings.py` under `INSTALLED_APPS`:

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

4. Enter the following code to create the models:

```
from django.db import models


class Project(models.Model):
    name = models.CharField(max_length=50,
    help_text="Project Name")
    creation_time = models.DateTimeField
    (auto_now_add=True,
    help_text="Project creation time.")
```

The full code can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter02/Activity2.01/juggler/projectm/models.py`.

5. Create the migration scripts, and migrate the models by executing the following commands separately:

```
python manage.py makemigrations
python manage.py migrate
```

6. Open the Django shell using the following command:

```
python manage.py shell
```

7. In the shell, add the following code to import the model classes:

```
>>> from projectm.models import Project, Task
```

8. Create a project while assigning the object to a variable:

```
>>> project = Project.objects.create(name='Paint the house')
```

9. Create tasks for the project:

```
>>> Task.objects.create(title='Paint kitchen',
description='Paint the kitchen', project=project, time_
estimate=4)


>>> Task.objects.create(title='Paint living room',
description='Paint the living room using beige color',
project=project, time_estimate=6)


>>> Task.objects.create(title='Paint other rooms',
description='Paint other rooms white', project=project, time_
estimate=10)
```

10. List all the tasks associated with the project:

```
>>> project.task_set.all()
<QuerySet [<Task: Buy paint and tools>, <Task: Paint kitchen>,
<Task: Paint living room>, <Task: Paint other rooms>]>
```

11. Using a different query method, list all the tasks associated with the project:

```
>>> Task.objects.filter(project__name='Paint the house')
<QuerySet [<Task: Buy paint and tools>, <Task: Paint kitchen>,
<Task: Paint living room>, <Task: Paint other rooms>]>
```

In this activity, given an application requirement, we created a project model, populated the database, and performed database queries.

Chapter 3, URL Mapping, Views, and Templates

Activity 3.01 – implementing the book details view

The following steps will help you complete this activity:

1. Create a file called `bookr/reviews/templates/reviews/book_detail.html` and add the following HTML code:

reviews/templates/reviews/book_detail.html

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

{% block content %}
      <br>
      <h3>Book Details</h3>
      <hr>
      <span class="text-info">Title: </span>
      <span>{{ book.title }}</span>
      <br>
      <span class="text-info">Publisher:
      </span><span>{{ book.publisher }}</span>
```

You can view the complete code at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter03/Activity3.01/bookr/reviews/templates/reviews/book_detail.html`.

Like the `books_list.html` template, this template also inherits from the `base.html` file by using the following code:

```
{% extends 'base.html' %}
{% block content %}
{% endblock %}
```

Because it inherited from `base.html`, you will be able to see the navigation bar on the book details page as well. The remaining part of the template uses the context to display the details of the book.

2. Open `bookr/reviews/views.py`, and append the view method as follows by retaining all the other code in the file as it is:

```
def book_detail(request, pk):
    book = get_object_or_404(Book, pk=pk)
    reviews = book.review_set.all()
    if reviews:
         book_rating = average_rating([review.rating
             for review in reviews])
```

```
            context = {"book": book,
                "book_rating": book_rating,
                "reviews": reviews}
    else:
        context = {"book": book,
            "book_rating": None,
            "reviews": None}
    return render(request, "reviews/book_detail.html",
        context)
```

You will also need to add the following `import` statement at the top of the file:

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

The `book_detail` function is the book details view. Here, `request` is the HTTP `request` object that will be passed to any view function upon invocation. The next parameter, `pk`, is the primary key or ID of the book. This will be part of the URL path that is invoked when we open the book's detail page. We have already added this code in `bookr/reviews/templates/reviews/books_list.html`:

```
<a class="btn btn-primary btn-sm active" role="button"
  aria-pressed="true" href="/book/{{ item.book.id }}/">Reviews</
a>
```

The preceding code snippet represents a button called **Reviews** on the books list page. Upon clicking this button, `/book/<id>` will be invoked. Note that `{{ item.book.id }}` refers to the ID or primary key of a book. The **Reviews** button should look like this:

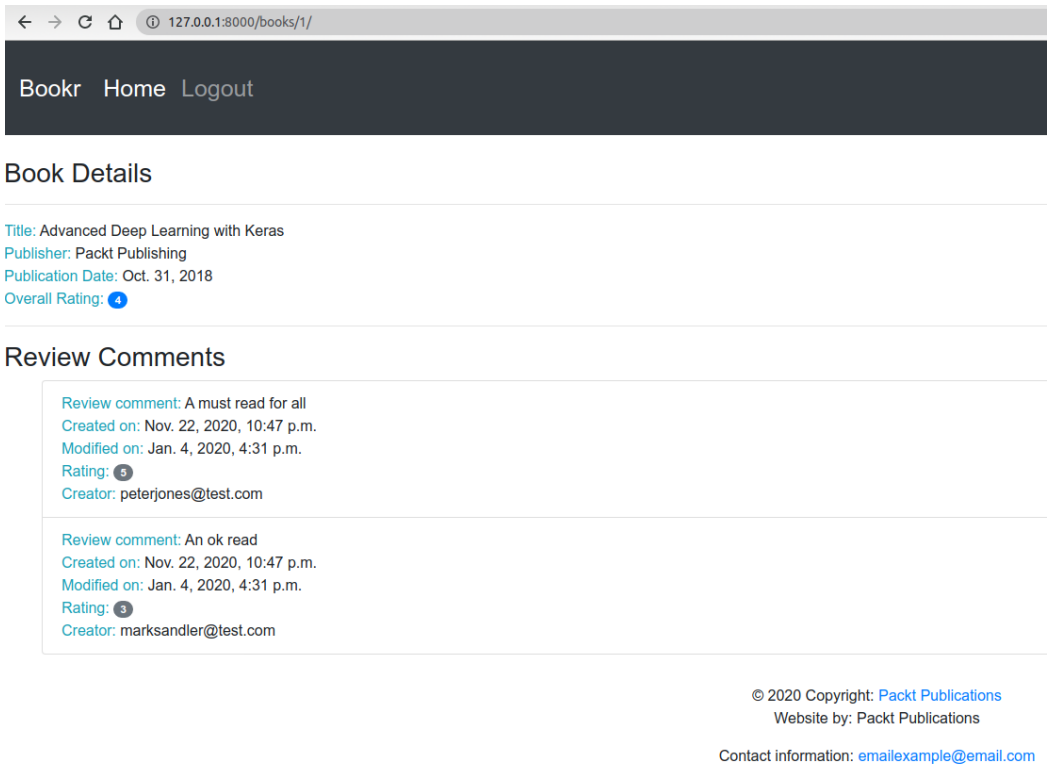
Activity Figure 3.1: The Reviews button below the details

3. Open `bookr/reviews/urls.py` and add a new path to the existing URL patterns as follows:

```
urlpatterns = [
    path('books/', views.book_list, name='book_list'),
    path('books/<int:pk>/', views.book_detail,
        name='book_detail')]
```

Here, `<int:pk>` represents an integer URL pattern. The newly added URL path is to identify and map the `/book/<id>/` URL of the book. Once it is matched, the `book_detail` view will be invoked.

4. Save all the modified files, and once the Django service restarts, open `http://0.0.0.0:8000/` or `http://127.0.0.1:8000/` in the browser to see the **Book Details** view with all the review comments:



Activity Figure 3.2: The page displaying the book details

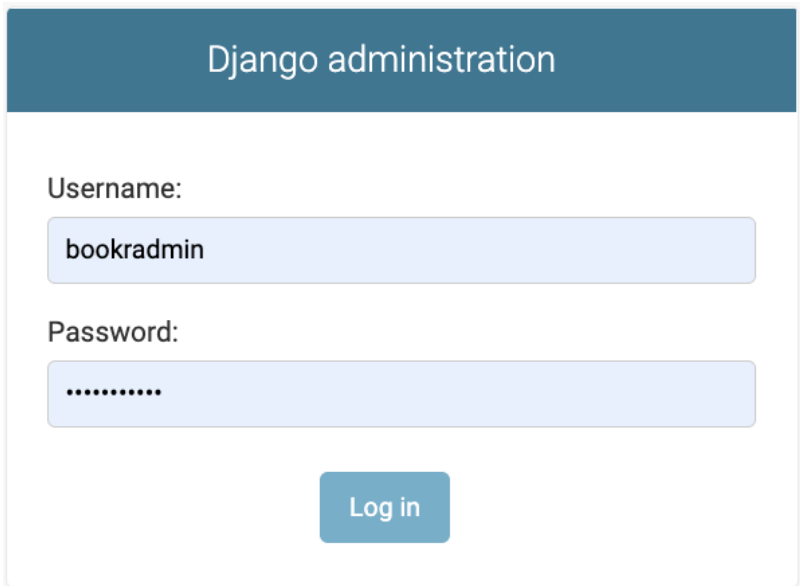
In this activity, we implemented the book details view, template, and URL mapping to display all the details of the book and its review comments.

Chapter 4, An Introduction to Django Admin

Activity 4.01 – creating a user account

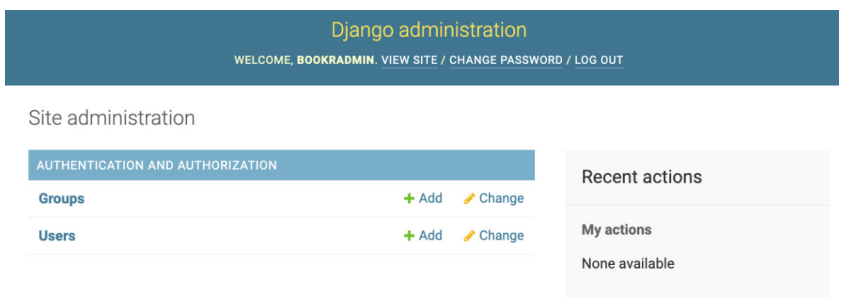
A user can be created with the following steps:

1. Visit `http://127.0.0.1:8000/admin/`. Log in to the admin app using the super-user account that you created in *Exercise 4.01*.



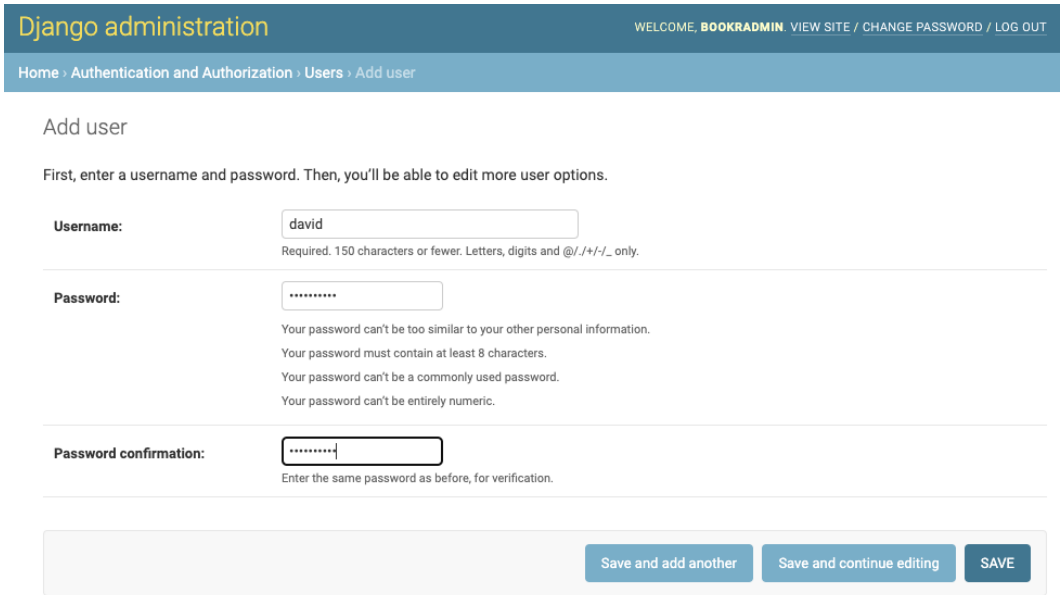
Activity Figure 4.1: The Django administration login form

2. Click on the **+ Add** link next to **Users**.



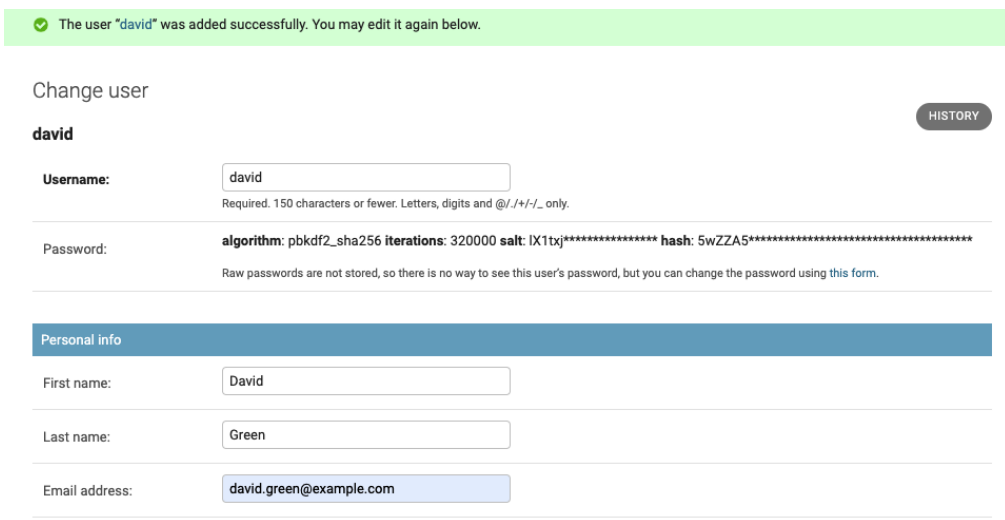
Activity Figure 4.2: The Django administration window

3. Enter a username and password. Then, click **Save and continue editing**.



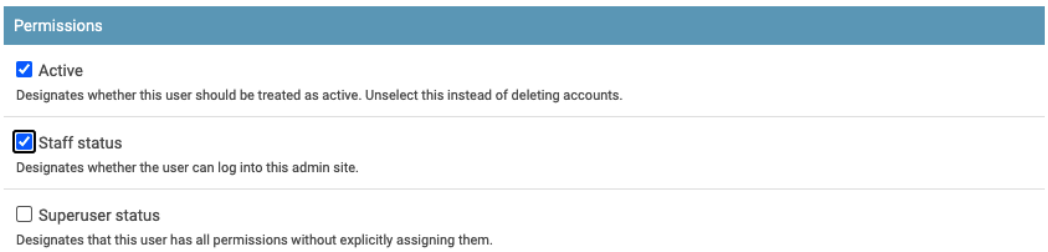
Activity Figure 4.3: The Add user page

4. The username and hashing information for the password are prefilled. You can now enter **First name**, **Last name**, and **Email address** details under **Personal info**.



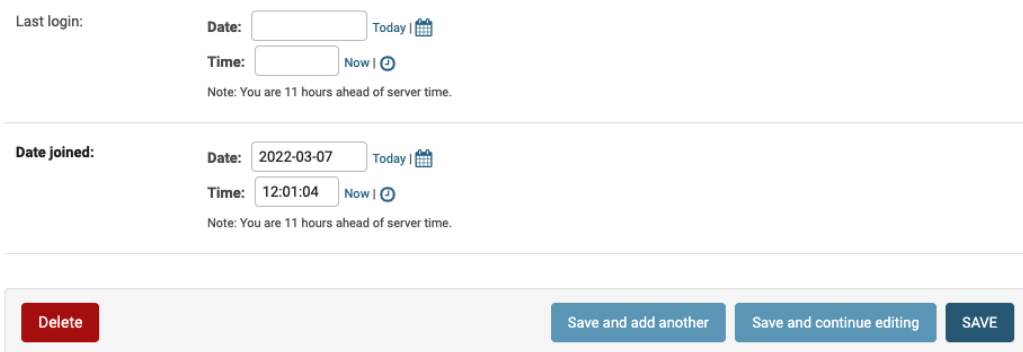
Activity Figure 4.4: The Change user page, presented after clicking Save and continue editing

5. Under **Permissions**, we are required to keep **Active** ticked, and also tick **Staff status**.



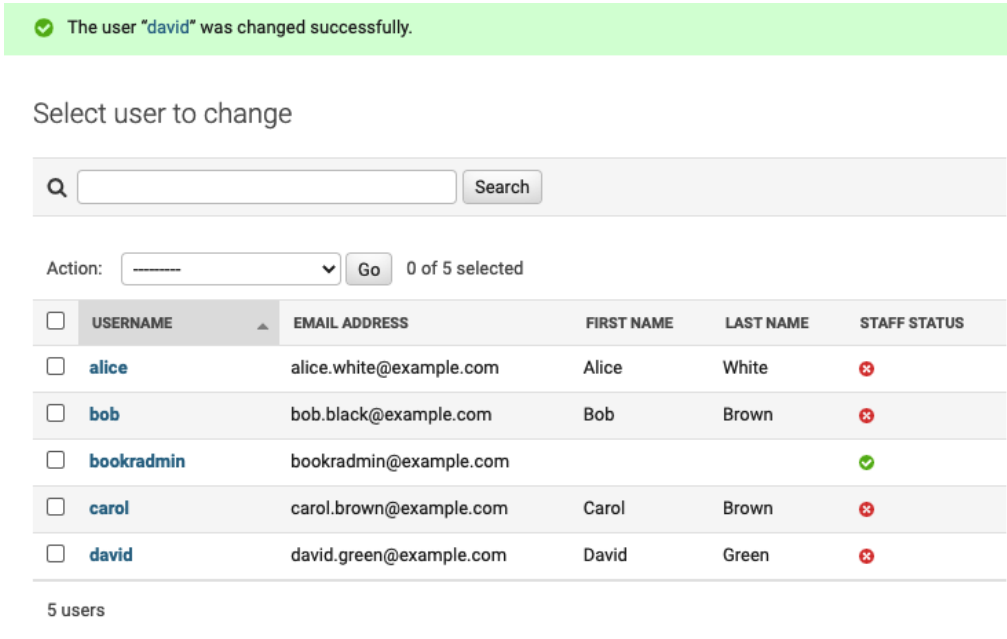
Activity Figure 4.5: User permissions options

6. Scroll to the bottom of the page and click **SAVE**.



Activity Figure 4.6: The Delete and SAVE buttons at the bottom of the User add form

When you complete the steps in this activity, you will be returned to the **User change** list page, where there is a success status message and a new listing for the created user.



Activity Figure 4.7: New list of the created users

Activity 4.02 – customizing the site admin

Let's complete the activity with the following steps:

1. Create the Django project app, run the migrations, create the super-user, and run the app:

```
django-admin startproject comment8or
cd comment8or/
python manage.py startapp messageboard
python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
  sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying
  admin.0003_logentry_add_action_flag_choices... OK
  Applying
  contenttypes.0002_remove_content_type_name... OK
```

```
  Applying
  auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying
  auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length...
  OK
  Applying
  auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying sessions.0001_initial... OK
python manage.py createsuperuser
Username (leave blank to use '<your system
username>'): c8admin
Email address: c8admin@example.com
Password:
Password (again):
Superuser created successfully.
python manage.py runserver
```

Copy the template from the Django source, `django/contrib/admin/templates/registration/logged_out.html`, to an appropriate path in the project directory, `comment8or/templates/comment8or/logged_out.html`.

2. Replace the `Thanks for spending some quality time with the Web site today.` text with `Bye from c8admin.`:

```
{% extends "admin/base_site.html" %}
{% load i18n %}

{% block breadcrumbs %}<div class="breadcrumbs">
  <a href="{% url 'admin:index' %}">
    {% trans 'Home' %}</a></div>{% endblock %}

{% block content %}

<p>{% trans "Bye from c8admin." %}</p>

<p><a href="{% url 'admin:index' %}">{% trans 'Log in
again' %}
  </a></p>
```

```
{% endblock %}
```

3. Add `admin.py` to the project directory with the following code:

```
from django.contrib import admin

class Comment8orAdminSite(admin.AdminSite):
    index_title = 'c8admin'
    title_header = 'c8 site admin'
    site_header = 'c8admin'
    logout_template = 'comment8or/logged_out.html'
```

4. Create a file called `messageboard/adminconfig.py` and add a custom `AdminConfig` subclass, as follows:

```
from django.contrib.admin.apps import AdminConfig

class MessageboardAdminConfig(AdminConfig):
    default_site = 'admin.Comment8orAdminSite'
```

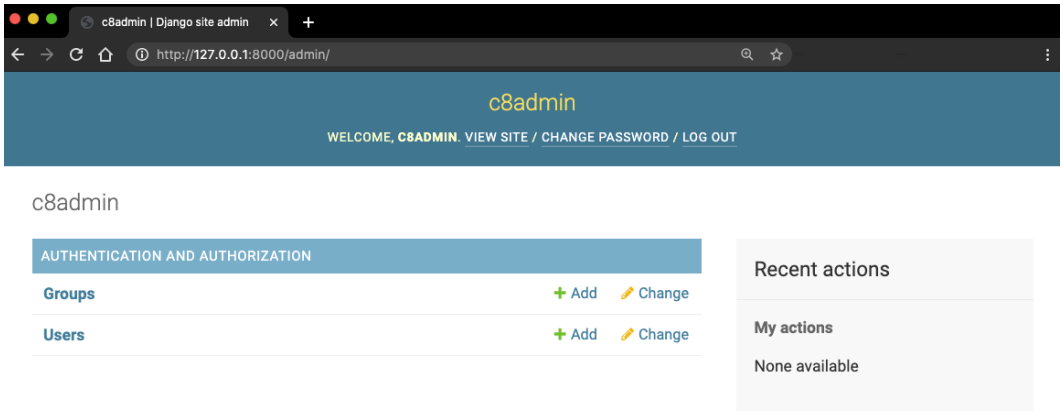
5. In the `INSTALLED_APPS` variable of `comment8or/settings.py`, replace the admin app with the custom `AdminConfig` subclass, `messageboard.adminconfig.MessageboardAdminConfig`, and add the `messageboard` app:

```
INSTALLED_APPS = [
   'messageboard.adminconfig.MessageboardAdminConfig',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'messageboard',]
```

6. Configure the `TEMPLATES` setting so that the project's template is discoverable in `comment8or/settings.py`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django
            .DjangoTemplates',
        'DIRS':
            [os.path.join(BASE_DIR,
            'comment8or/templates')],
            …}]
```

By completing this activity, you have created a new project and successfully customized the admin app by sub-classing `AdminSite`. The customized admin app in your `Comment8r` project should look something like this:



Activity Figure 4.8: The app index page after customization

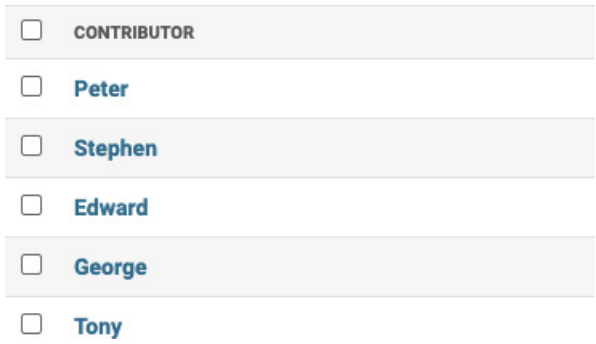
Activity 4.03 – customizing the model admins

The following steps will help you complete this activity:

1. The default list display for the `Contributors` change list uses the `Model.__str__` method's representation of a class name followed by `id`, such as `Contributor (12)`, `Contributor(13)`, and so on, as follows:

Activity Figure 4.9: The default display for the Contributors change list

In *Chapter 2*, *Models and Migrations*, a `__str__` method is added to the `Contributor` model so that each contributor is represented by `first_names`:



Activity Figure 4.10: Contributors represented by first names

These steps are to create a more intuitive string representation of the contributor, such as `Salinger, JD` instead of `Contributor (12)`.

2. Edit `reviews/models.py` and add an appropriate `initialled_name` method to `Contributor`.

   This code gets a string of initials of the contributor's `First Names` and then returns a string with the `Last Names`, followed by the initials:

```
    def initialled_name(self):
        """ self.first_names='Jerome David',
            self.last_names='Salinger'
            => 'Salinger, JD' """
        initials = ''.join([name[0] for name
            in self.first_names.split(' ')])
        return "{}, {}".format(self.last_names,
            initials)
```

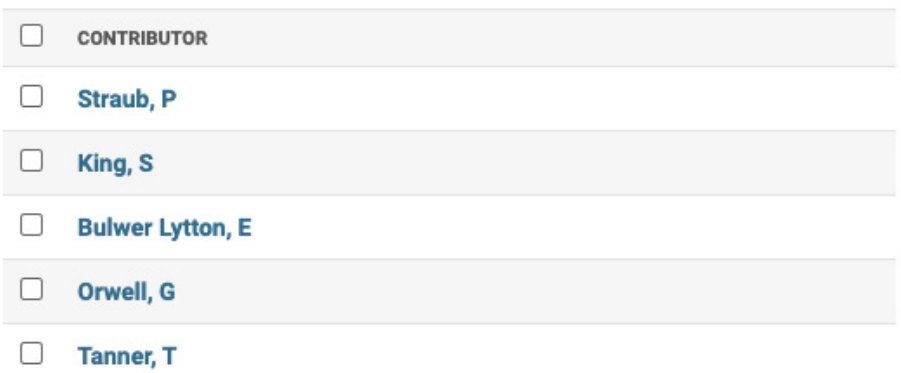
3. Replace the `__str__` method for `Contributor` with one that calls `initialled_name()`:

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

After these steps, the `Contributor` class in `reviews/models.py` will look like this:

```
class Contributor(models.Model):
    """A contributor to a Book, e.g. author, editor,
    co-author."""
    first_names = models.CharField(max_length=50,
        help_text="The contributor's first name or
        names.")
    last_names = models.CharField(max_length=50,
        help_text="The contributor's last name or
        names.")
    email = models.EmailField(
        help_text="The contact email for the
        contributor.")

    def initialled_name(self):
        """ self.first_names='Jerome David',
            self.last_names='Salinger'
            => 'Salinger, JD' """
        initials = ''.join([name[0] for name
            in self.first_names.split(' ')])
        return "{}, {}".format(self.last_names,
            initials)

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

The `Contributors` change list will appear as follows:



Activity Figure 4.11: The Contributors change list with the last name and initial of the first name

4. Add a `ContributorAdmin` class that inherits from `admin.ModelAdmin`:

```
class ContributorAdmin(admin.ModelAdmin):
```

5. Modify it so that in the `Contributors` change list, records are displayed with two sortable columns – `Last Names` and `First Names`:

```
    list_display = ('last_names', 'first_names')
```

6. Add a search bar that searches on `Last Names` and `First Names`. Modify it so that it only matches the start of `Last Names`:

```
    search_fields = ('last_names__startswith',
    'first_names')
```

7. Add a filter on `Last Names`:

```
    list_filter = ('last_names',)
```

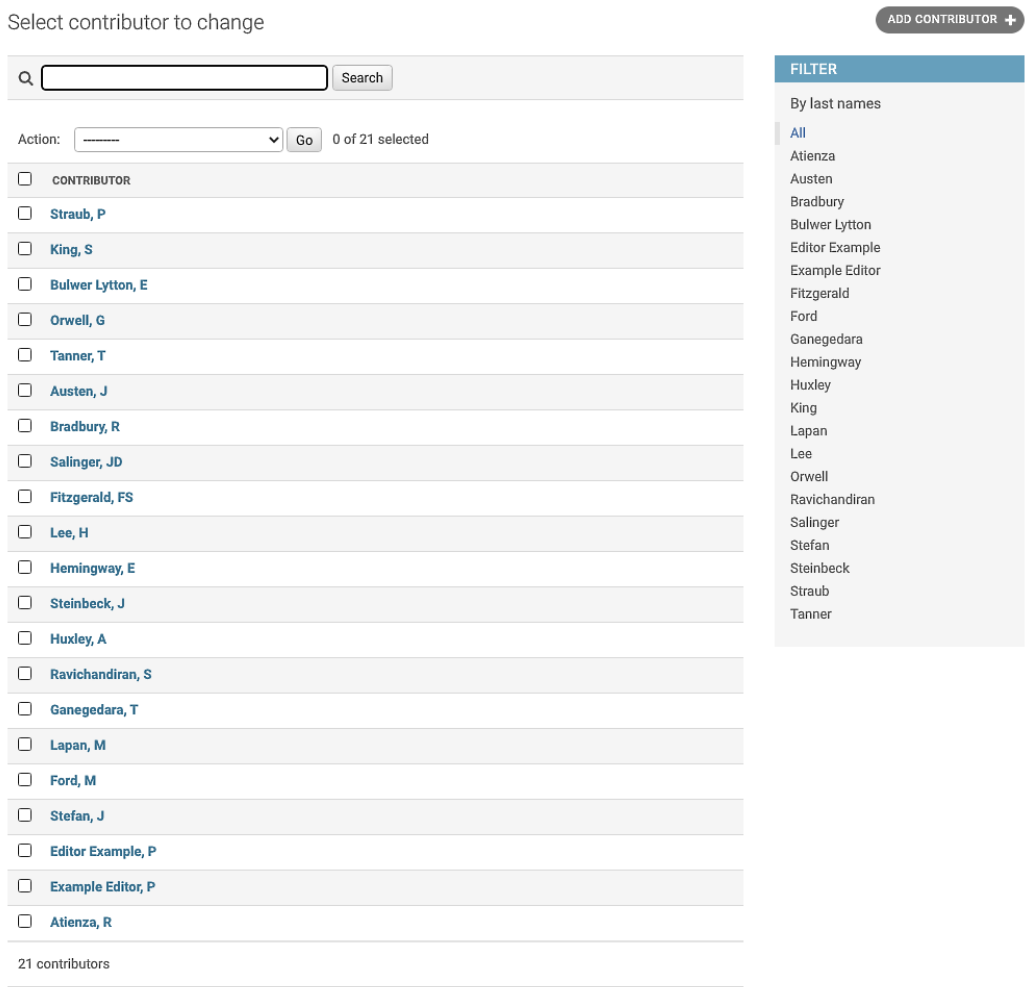
8. Modify the `register` statement for the `Contributor` class to include the `ContributorAdmin` argument:

```
admin.site.register(Contributor, ContributorAdmin)
```

The changes to `reviews/admin.py` should look like this:

```
class ContributorAdmin(admin.ModelAdmin):
    list_display = ('last_names', 'first_names')
    list_filter = ('last_names',)
    search_fields = ('last_names__startswith',
    'first_names')


admin.site.register(Contributor, ContributorAdmin)
```

With the two sortable columns, the filter, and the search bar, the list will look like this:



Activity Figure 4.12: The completed contributors change list

Chapter 5, Serving Static Files

Activity 5.01 – adding a reviews logo

Perform the following steps to complete this activity:

1. Open `base.html` in the main `templates` directory. Find the `<style>` tags in `<head>`, and add this rule at the end (after the `.navbar-brand` rule):

```
.navbar-brand > img {
  height: 60px;
}
```

After adding this rule to your `<style>` element, it should look like *Activity Figure 5.1*.

```
<style type="text/css">
  .navbar {
    min-height: 100px;
    font-size: 25px;
  }
  .navbar-brand {
    font-size: 25px;
  }

  .navbar-brand > img {
    height: 60px;
  }
</style>
```

Activity Figure 5.1: A new rule added to the <style> element

2. Locate the following line:

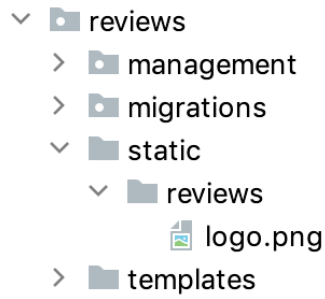
```
<a class="navbar-brand" href="/">Book Review</a>
```

It will be just inside the start of `<body>`. Change it to add the `{% block brand %}` and `{% endblock %}` wrappers around the text:

```
<a class="navbar-brand" href="/">{% block brand %}Book
Review{% endblock %}</a>
```

You can save and close `base.html`.

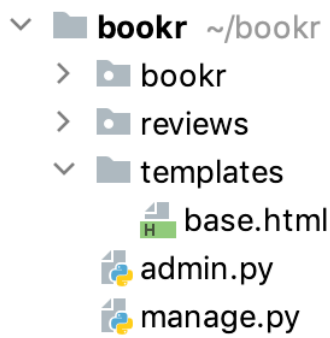
3. Create a directory named `static`, inside the `reviews` app directory. Inside this `static` directory, create a directory named `reviews`. Put `logo.png` from `https://raw.githubusercontent.com/PacktPublishing/Web-Development-with-Django-Second-Edition/main/Chapter05/Activity5.01/bookr/reviews/static/reviews/logo.png` inside this directory. Your final directory structure should look like this:



Activity Figure 5.2: A layout of the reviews static directory with logo.png

4. Create a `templates` directory inside the Bookr project directory. Move the `reviews/templates/reviews/base.html` file into this directory.

   Your directory structure should look like this:



Activity Figure 5.3: The layout of the project templates directory

5. Open `settings.py` and locate the `TEMPLATES = ...` setting. Add `BASE_DIR / "templates"` to the `DIRS` setting so that the `TEMPLATES` setting looks like this:

```
TEMPLATES = [
    {
        "BACKEND":
            "django.template.backends.django
            .DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
               "django.template.context_processors
                   .debug",
               "django.template.context_processors
                   .request",
               "django.contrib.auth.context_processors
                   .auth",
               "django.contrib.messages.context
                   _processors.messages",
            ],
        },
    },
]
```

   Save and close `settings.py`.

6. Right-click on the `reviews` namespaced template directory and choose **New | File** (not **New | HTML File**, as we don't require the HTML boilerplate that would generate). Name the file `base.html`.

   The file will open. Add an `extends` template tag to the file, to extend from `base.html`:

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

7. We will use the `{% static %}` template tag to generate the `img` URL, so we need to make sure the `static` library is loaded. Add this line after the `extends` line:

```
{% load static %}
```

   Then, override the `{% block brand %}` content, and use the `{% static %}` template tag to generate the URL to the `reviews/logo.png` file:

```
{% block brand %}<img src="{% static
'reviews/logo.png' %}">{% endblock %}
```

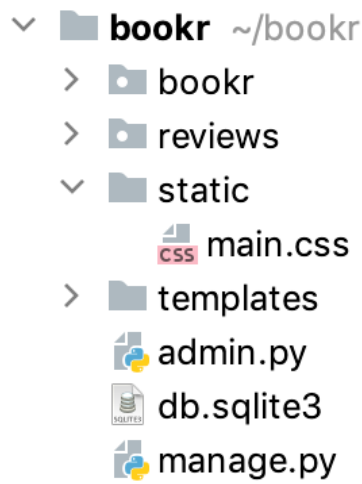
There should now be three lines in this new `base.html` file. You can save and close it. The completed file is available at `https://raw.githubusercontent.com/PacktPublishing/Web-Development-with-Django-Second-Edition/main/Chapter05/Activity5.01/bookr/reviews/templates/reviews/base.html`.

Start the Django dev server, if it's not already running. The pages you should check are the home page, which should look the same. Then, check the books list page and book details page, which should have the Bookr Reviews logo displayed.

Activity 5.02 – CSS enhancements

Perform the following steps to complete this activity:

1. In PyCharm, right-click on the `bookr` project directory and select **New | Directory**. Name the directory `static`. Then, right-click on it and select **New | File**. Name the file `main.css`. Your directory layout should then look like *Activity Figure 5.4*.



Activity Figure 5.4: The main.css file added

2. Open the main `base.html` file, and then find the `style` element in `head`. Copy its contents (but not the `<style>` and `</style>` tags themselves) into the `main.css` file. Then, at the end, add the CSS snippet given in the activity instructions. Your file should contain this content when finished:

```
.navbar {
  min-height: 100px;
  font-size: 25px;
}
```

```
.navbar-brand {
  font-size: 25px;
}

.navbar-brand > img {
  height: 60px;
}

body {
  font-family: 'Source Sans Pro', sans-serif;
  background-color: #e6efe8;
  color: #393939;
}

h1, h2, h3, h4, h5, h6 {
  font-family: 'Libre Baskerville', serif;
}
```

You can save and close `main.css`. Then, return to `base.html` and remove the entire `style` element.

The complete `main.css` can be found at `https://github.com/PacktWorkshops/The-Django-Workshop/blob/master/Chapter05/Activity5.02/bookr/static/main.css`.

3. Load the `static` library on the second line of `base.html`, by adding this:

```
{% load static %}
```

Then, you can use the `{% static %}` tag to generate the URL for `main.css`, to be used in a `link` element:

```
<link rel="stylesheet" href="{% static 'main.css' %}">
```

Insert this into the `head` of the page, where `<style>` was before you removed it.

4. Underneath `<link>` you added in the previous step, add the following to include the Google Fonts CSS:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/
css?family=Libre+Baskerville|Source+Sans+Pro&display=swap">
```

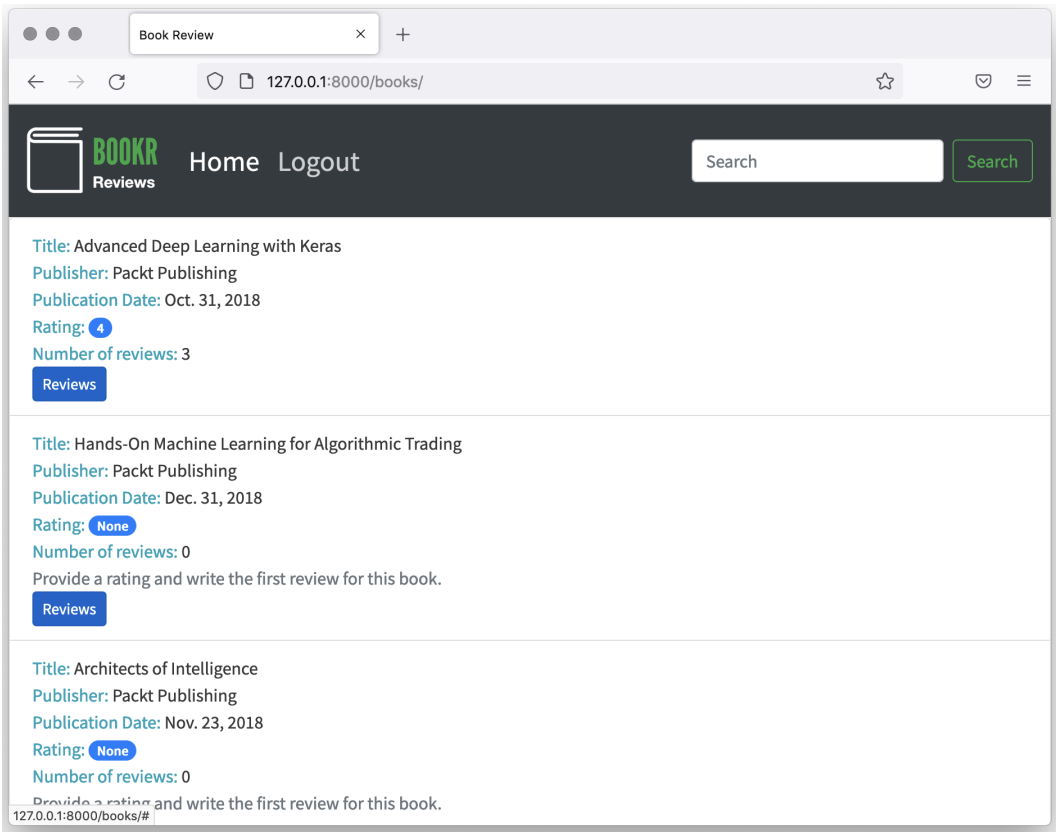
You can now save and close `base.html`. The completed file can be found at `https://github.com/PacktWorkshops/The-Django-Workshop/blob/master/Chapter05/Activity5.02/bookr/templates/base.html`.

5. Open `settings.py` in the `bookr` package directory. Scroll to the bottom of the file and add this line:

```
STATICFILES_DIRS = [BASE_DIR / "static"]
```

This will set the `STATICFILES_DIRS` setting to a single-element list, containing the path to the `static` directory in your project. This will allow Django to locate the `main.css` file. The completed `settings.py` can be found at `https://github.com/PacktWorkshops/The-Django-Workshop/blob/master/Chapter05/Activity5.02/bookr/bookr/settings.py`.

6. Start the Django dev server – you may need to restart it to load the changes to `settings.py`. Then, visit `http://127.0.0.1:8000/` in your browser. You should see updated fonts and background colors on all the Bookr pages:

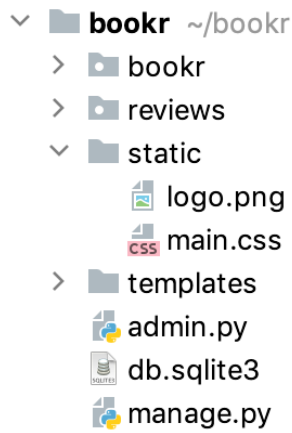


Activity Figure 5.5: The book list with new font

Activity 5.03 – adding a global logo

Perform the following steps to complete this activity:

1. Download the `logo.png` file from `https://raw.githubusercontent.com/PacktPublishing/Web-Development-with-Django-Second-Edition/main/Chapter05/Activity5.03/bookr/static/logo.png`.
2. Move the logo into the project `static` directory (not the `reviews` `static` directory).



Activity Figure 5.6: The Bookr project directory layout

Figure 5.6 shows the project pane in PyCharm illustrating the correct location for `logo.png`.

3. Open `templates/base.html` (not `reviews/templates/reviews/base.html`). Locate the `{% block brand %}` template tag (inside the `<a class="navbar-brand">` tag). It will have the `Book Review` content, like this:

```
{% block brand %}Book Review{% endblock %}
```

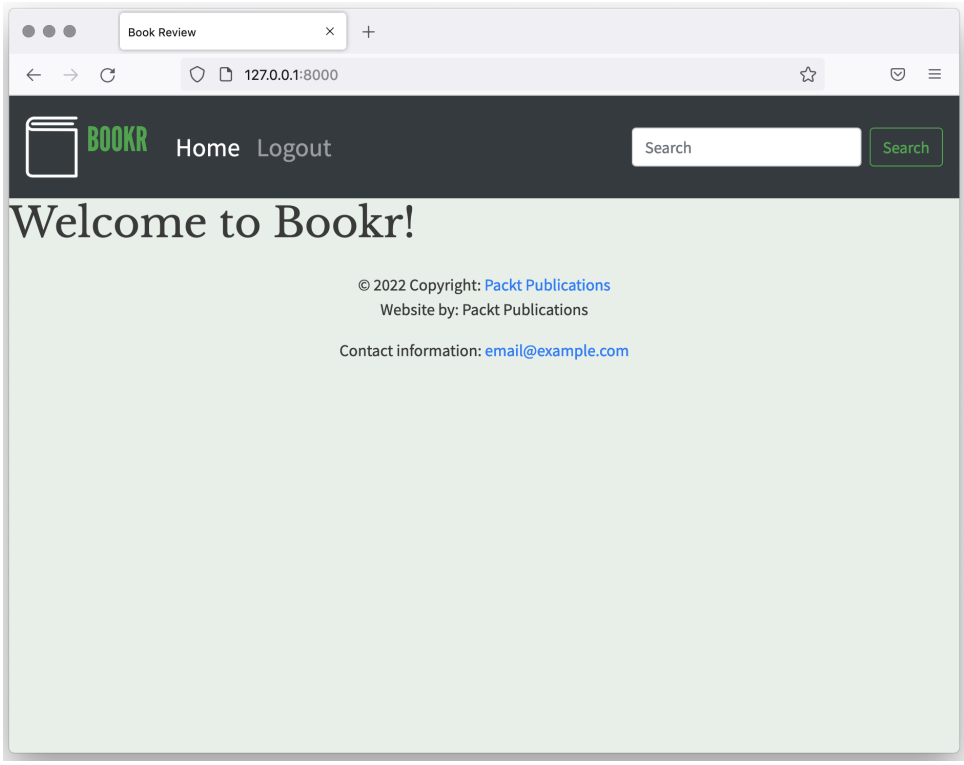
Change the content to an `<img>` tag. The `src` attribute should use the `static` template tag to generate the URL of `logo.png`, like this:

```
{% block brand %}<img src="{% static 'logo.png' %}">{%
endblock %}
```

You have already loaded the `static` template tag library (by adding the `load` template tag) in *Activity 5.01*.

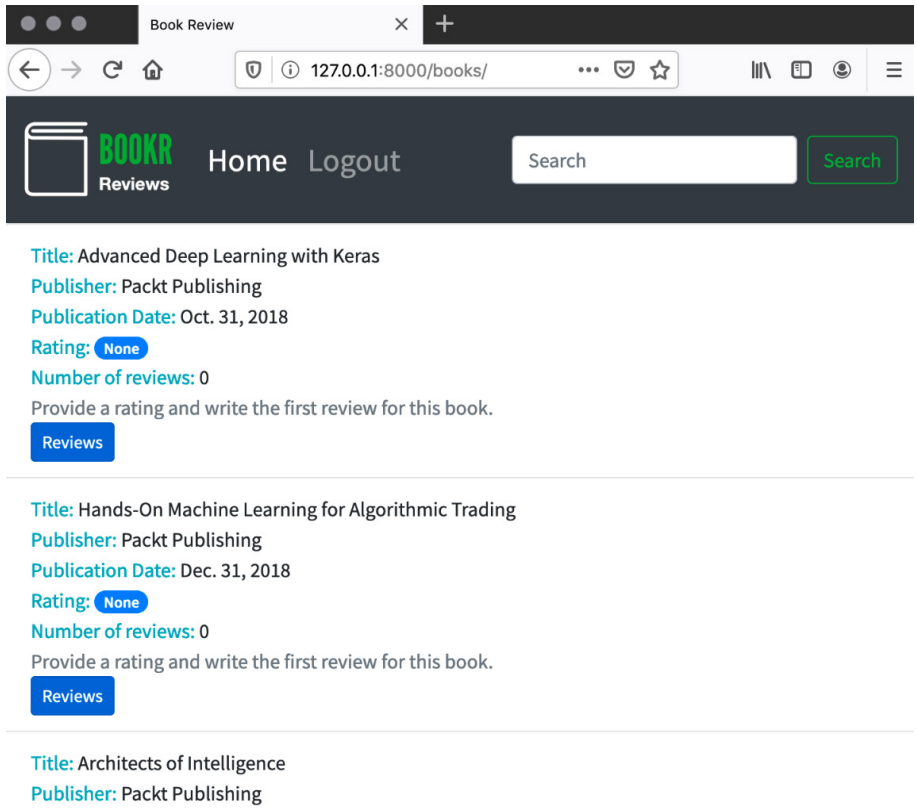
Note that like `main.css`, `logo.png` is not namespaced, so we don't need to include a directory name.

4. Start the Django development server if it is not already running, and then open `http://127.0.0.1:8000/` in your browser. You should see that the main Bookr page now has the Bookr logo, as shown in the following figure.



Activity Figure 5.7: The Bookr Logo on the main page

In contrast, if you open a reviews page, such as `http://127.0.0.1:8000/books/`, you'll see the Bookr Reviews logo as you did previously. Refer to *Activity Figure 5.8* for how it should look.



Activity Figure 5.8: The Bookr Reviews logo still shows on the Reviews pages

In this activity, you added a global logo in the base template (`base.html`). The logo shows on all non-review pages, but the existing reviews logo still shows on the review pages.

You saw that with the use of namespacing, we can refer to each `logo.png` by the directory in which it resides (`logo.png` for the global logo and `reviews/logo.png` for the review-specific logo). You used the `static` template tag to generate the URL to the logo file. This will allow us flexibility when deploying to production and gives us the option to easily change the static URL by updating the `STATIC_URL` setting.

Chapter 6, Forms

Activity 6.01 – book searching

Perform the following steps to complete this activity:

1. Open `forms.py` in the `reviews` app. Create a new class called `SearchForm` that inherits from `forms.Form`. The class definition should look like this:

```
class SearchForm(forms.Form):
```

2. Add a `CharField` instance called `search`, which is instantiated with the `required` argument set to `False` and the `min_length` argument set to `3`:

```
class SearchForm(forms.Form):
    search = forms.CharField(required=False,
    min_length=3)
```

This will ensure that the field does not need to be filled in, but if data is entered, it must be at least three characters long before the form is valid.

3. Add a `ChoiceField` instance named `search_in`. It should also be instantiated with the `required` argument set to `False`. The `choices` argument should be a tuple of tuples in the form (`value`, `description`):

```
(("title", "Title"), ("contributor", "Contributor"))
```

Note that this could be a list of lists, a tuple of tuples, or any combination of the two – this is also valid:

```
[("title", "Title"), ["contributor", "Contributor"]]
```

Ideally, though, you would be consistent in your choice of objects – that is, all tuples or all lists.

After completing *steps 1–3*, your completed `SearchForm` class should look like this:

```
class SearchForm(forms.Form):
    search = forms.CharField(required=False,
        min_length=3)
    search_in = forms.ChoiceField(required=False,
        choices=(("title", "Title"),
        ("contributor",
        "Contributor")))
```

4. Open the `reviews` app's `views.py` file. First, make sure `SearchForm` is imported. You will already be importing `ExampleForm`, so just add `SearchForm` to the `import` line. Take the following line:

```
from .forms import ExampleForm
```

Change it to this:

```
from .forms import ExampleForm, SearchForm
```

Then, inside the `book_search` view, instantiate a `SearchForm` instance, passing in `request.GET`. Assign this to the `form` variable:

```
def book_search(request):
    search_text = request.GET.get("search", "")
    form = SearchForm(request.GET)
```

5. In the `book_search` view, you will need to add a placeholder `books` empty `set` variable, which will be used in the `render` context if the form is not valid. We will also use it to build the results in the next step:

```
def book_search(request):
    # Code from Step 4 truncated
    books = set()
```

Then, we should only proceed with a search if the form is valid and some search text has been entered (remember, the form is valid even if the search is empty).

6. We will then check whether the form's `cleaned_in` value is `"title"` and, if so, filter the `Book` objects using the `title__icontains` argument to perform a case-insensitive search. Putting this all together, the `book_search` view up to this point should look like this:

```
def book_search(request):
    # Code from Step 4/6 truncated

    if form.is_valid() and
    form.cleaned_data["search"]:
        search = form.cleaned_data["search"]
        search_in = form.cleaned_data.get("search_in")
            or "title"
        if search_in == "title":
            books = Book.objects.filter
                (title__icontains=search)
```

7. Since we are going to search `Contributors`, make sure to import `Contributor` at the start of the file. Find the following line:

```
from .models import Book
```

Change it to this:

```
from .models import Book, Contributor
```

If the `search_in` value of `SearchForm` is not `title`, then it must be `"contributor"` (since there are only two options). An `else` branch can be added to the last `if` statement added in the previous step, which searches by `Contributor`.

You might want to search by `Contributor` by passing both arguments to the same `filter` call, as shown here:

```
contributors = Contributor.objects.filter(
    first_names__icontains=search,
    last_names__icontains=search)
```

However, in this case, Django will perform this as an `AND` search, so the search term would need to be present for the `first_names` and `last_names` values of the same author.

Instead, we will perform two queries and iterate them separately. Every contributor that matches has each of its `Book` instances added to the `books` set that was created in the previous step:

```
fname_contributors =
    Contributor.objects.filter(first_names__icontains=
        search)

for contributor in fname_contributors:
    for book in contributor.book_set.all():
        books.add(book)

lname_contributors =
    Contributor.objects.filter(last_names__icontains=
        search)

for contributor in lname_contributors:
    for book in contributor.book_set.all():
        books.add(book)
```

Since `books` is a `set` instance instead of a `list` instance, duplicate `Book` instances are avoided automatically.

Note that you could also convert the query results into lists by passing them to the `list` constructor function and then combining them with the + operator. Then, just iterate over the single combined list:

```
contributors = list(Contributor.objects.filter
    (first_names__icontains=search)) +
        list(Contributor.objects.filter
            (last_names__icontains=search))

for contributor in contributors:
    for book in contributor.book_set.all():
        books.add(book)
```

This is still not ideal, as we make two separate database queries. Instead, you can combine queries with an `OR` operator using the | (pipe) character. This will make just one database query:

```
contributors = Contributor.objects.filter
    (first_names__icontains=search) |
```

```
        Contributor.objects.filter
            (last_names__icontains=search)
```

Then, again, only one `contributor` iterating loop is required:

```
for contributor in contributors:
    for book in contributor.book_set.all():
        books.add(book)
```

The `book_search` view should look like this at this point:

```
def book_search(request):
    search_text = request.GET.get("search", "")
    form = SearchForm(request.GET)

    books = set()

    if form.is_valid() and
    form.cleaned_data["search"]:
        search = form.cleaned_data["search"]
        search_in = form.cleaned_data.get("search_in")
            or "title"
        if search_in == "title":
            books = Book.objects.filter
            (title__icontains=search)
        else:
            fname_contributors =

                Contributor.objects.filter
                (first_names__icontains=search)

                for contributor in fname_contributors:
                    for book in
                    contributor.book_set.all():
                        books.add(book)

            lname_contributors =
                Contributor.objects.filter
                (last_names__icontains=search)

            for contributor in lname_contributors:
                for book in
                contributor.book_set.all():
                books.add(book)
```

8. The context dictionary being passed to `render` should include the `search_text` variable, the `form` variable, and the `books` variable – this might be an empty set. For simplicity, the keys can match the values. The second argument to render should include the `reviews` directory in the template path. The `render` call should look like this:

```
return render(request, "reviews/search-results.html",
    {"form": form, "search_text": search_text,
    "books": books})
```

9. Open `search-results.html` inside the `reviews` templates directory. Delete all its contents (since we created it before template inheritance was covered, its previous content is now redundant). Add an `extends` template tag at the start of the file, to extend from `base.html`:

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

Under the `extends` template tag, add a `block` template tag for the `title` block. Add an `if` template tag that checks whether `form` is valid and whether `search_text` is set – if so, render `Search Results for "{{ search_text }}"`. Otherwise, just render the static `Book Search` text. Remember to close the block with an `endblock` template tag. In the context of the `extends` template tag, your file should now look like this:

```
{% extends 'base.html'%}
{% block title %}
    {% if form.is_valid and search_text %}
        Search Results for "{{ search_text }}"
    {% else %}
        Book Search
    {% endif %}
{% endblock %}
```

10. After the title's `endblock` template tag, add the opening `content block` template tag:

```
{% block content %}
```

Add the `<h2>` element with the static `Search for Books` text:

```
<h2>Search for Books</h2>
```

Then, add a `<form>` element and render the form inside using the `as_p` method:

```
<form>
    {{ form.as_p }}
```

Your `<button>` element should be similar to the submit buttons you added in *Activity 6.01 – book searching*, with `submit` as `type` and `btn btn-primary` for `class`. Its text content should be `Search`:

```
<button type="submit" class="btn btn-primary">Search</button>
```

Finally, close the form with a `</form>` tag.

Note that we don't need `{% csrf_token %}` in this form, since we're submitting it using `GET` rather than `POST`. Also, we will close the block with an `endblock` template tag in *step 12*.

11. Under the `</form>` tag, add an `if` template tag that checks whether the form is valid and whether `search_text` has a value:

```
{% if form.is_valid and search_text %}
```

As in the view, we need to check both of these because the form is valid even if `search_text` is blank. Add the `<h3>` element underneath:

```
<h3>Search Results for <em>{{ search_text }}</em></h3>
```

We won't close the `if` template tag yet, as we use the same logic to show or hide the results, so we will close it in the next step.

12. First, under `<h3>`, add an opening `<ul>` tag, with a `list-group` class:

```
<ul class="list-group">
```

Then, add the `for` template tag to iterate over the `books` variable:

```
{% for book in books %}
```

Each book will be displayed in an `<li>` instance with a `list-group-item` class. Use a `span` instance with a `text-info` class, like what was used in `book_list.html`. Generate the URL of the link using the `url` template tag. The link text should be the `book` title:

```
<li class="list-group-item">
  <span class="text-info">Title:
    </span> <a href="{% url 'book_detail' book.pk %}">
      {{ book }}</a>
```

Use a `<br>` element to put the contributors on a new line, and then add another `span` with the `text-info` class to show the `Contributors` leading text:

```
<br/>
<span class="text-info">Contributors: </span>
```

Iterate over each contributor for the book (`book.contributors.all`), using a `for` template tag, and display their `first_names` and `last_names` values. Separate each contributor with a comma (use the `forloop.last` special variable to exclude a trailing comma):

```
{% for contributor in book.contributors.all %}
{{ contributor.first_names }} {{
contributor.last_names }}
{% if not forloop.last %}, {% endif %}
{% endfor %}
```

Close the `<li>` element with a closing `</li>` tag:

```
</li>
```

We then will display the message if there are no results, using the `empty` template tag, and then close the `for` template tag:

```
{% empty %}
<li class="list-group-item">No results found.</li>
{% endfor %}
```

Finally, we'll close all the HTML tags and template tags that we opened from *step 11* until now – first, the `</ul>` results, then the `if` template tag that checks whether we have results, and finally, the `content` block:

```
</ul>
{% endif %}
{% endblock %}
```

13. Open the project `base.html` template (not the `reviews` app's `base.html` template). Find the opening `<form>` tag (there is only one in the file) and set its `action` attribute to the URL of the `book_search` view, using the `url` template tag to generate it. After updating this tag, it should look like this:

```
<form action="{% url 'book_search' %}" class="form-
inline my-2 my-lg-0">
```

Since we are submitting as `GET`, we don't need to set a method attribute or include `{% csrf_token %}` in the `<form>` body.

14. The final steps are to set the name of the search `<input>` as `search`. Then, display the value of `search_text` in the `value` attribute. This is done using standard variable interpolation. Also, add a `minlength` attribute, set to `3`.

After updating, your `<input>` tag should look like this:

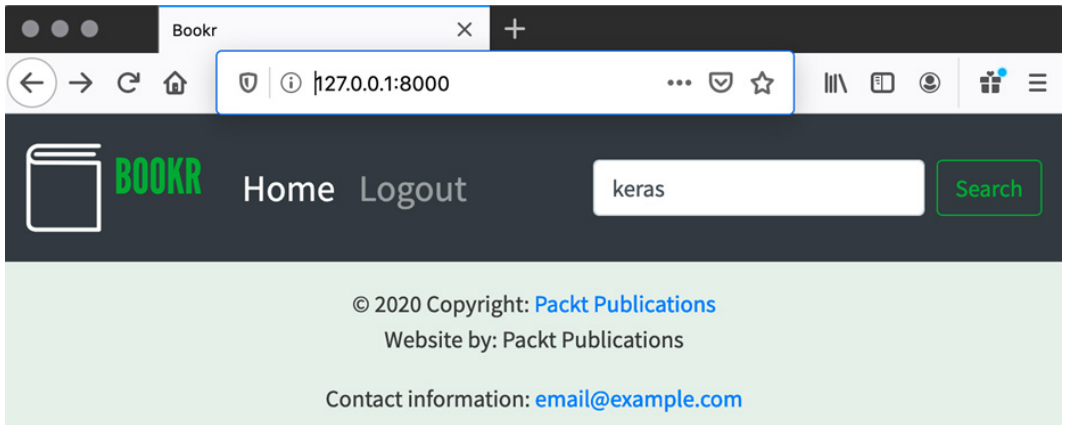
```
<input class="form-control mr-sm-2" type="search"
  placeholder="Search" aria-label="Search"
  name="search" value="{{ search_text }}"
  minlength="3">
```

This will display the search text on the search page in the top-right search field. On other pages, where there is no search text, the field will be blank.

15. Locate the `<title>` element (it should be just before the closing `</head>` tag). Inside it, add a `{% block title %}` instance with the text `Bookr`. Make sure to include the `{% endblock %}` closing template tag:

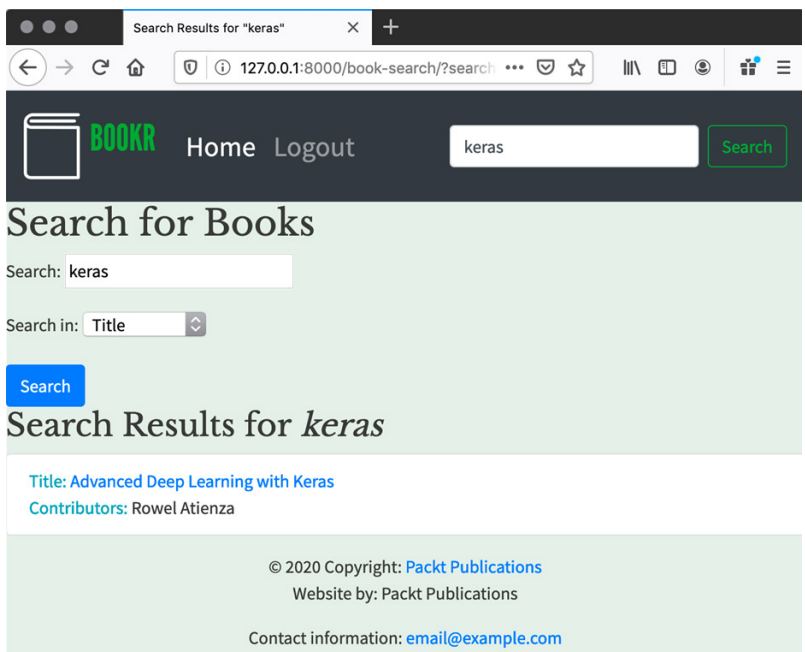
```
<title>{% block title %}Bookr{% endblock %}</title>
```

Start the Django development server if it is not already running. You can visit the main page at `http://127.0.0.1:8000/` and perform a search from there, and you will be taken to the search results page:



Activity Figure 6.1: The search text entered on the main page

We are still taken to the search results page to see the results (*Activity Figure 6.2*):



Activity Figure 6.2: The search results (for the title) are the same, regardless of which search field was used

Clicking the title link will take you to the book details page for the book.

Chapter 7, Advanced Form Validation and Model Forms

Activity 7.01 – styling and integrating the publisher form

The following steps will help you complete this activity:

1. In `templates/base.html`, locate the `{% block content %}` line. On the line before this, add the opening `<div class="container-fluid">` tag. Add a closing `</div>` after the corresponding `{% endblock %}`. This section should now look like this:

```
<div class="container-fluid">
  {% block content %}
    <h1>Welcome to Bookr!</h1>
  {% endblock %}
</div>
```

2. Add a `{% for %}` block to iterate over the `messages` variable. The loop should contain the snippet, as shown in *step 2* of the activity brief. It should be inside the `<div>` element added in *step 1* of the solution but before `{% block content %}`. The whole container `div` code should be like this:

```
<div class="container-fluid">
  {% for message in messages %}
    <div class="alert alert-{% if message.level_tag
    == 'error' %}danger{% else %}{{ message.level_tag
    }}{% endif %}"
  role="alert">
    {{ message }}
  </div>
  {% endfor %}
{% block content %}
  <h1>Welcome to Bookr!</h1>
{% endblock %}
</div>
```

3. Create a new file inside the `reviews/templates/reviews` directory:

Activity Figure 7.1: Creating a new file inside the reviews templates directory

There is no need to select the **HTML File** option, as we do not need the automatically generated content. Just selecting **File** is fine.

Name the file `instance-form.html`.

4. To `instance-form.html`, add an `extends` template tag. The template should extend from `reviews/base.html`, like this:

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

5. In *step 13*, you will render this template with the `form`, `instance`, and `model_type` context variables. You can write the code in the template to use them already though. Add the `{% block title %}` and `{% endblock %}` template tags. Between them, implement the logic to change the text based on `instance` being `None` or not:

```
{% block title %}
{% if instance %}
  Editing {{ model_type }} {{ instance }}
{% else %}
  New {{ model_type }}
{% endif %}
{% endblock %}
```

6. Add a new `{% block content %}` after the `{% endblock %}` added in *step 5*.
7. Add an `<h2>` element after `{% block content %}`. You can reuse the logic from *step 5*; however, wrap the `{{ instance }}` display in an `<em>` element:

```
{% endblock %} {# from step 5 #}

{% block content %}
<h2>
  {% if instance %}
    Editing {{ model_type }} <em>{{ instance }}</em>
  {% else %}
    New {{ model_type }}
  {% endif %}
</h2>
```

8. After the closing `</h2>` tag, add a pair of empty `<form>` elements. The `method` attribute should be `post`:

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

9. Add `{% csrf_token %}` to the line after the opening `<form>` tag.

10. Render the form using the {{ form.as_p }} tag. Do this inside the <form> element, after {% csrf_token %}. The code will look like this:

```
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
```

11. After the code added in *step 10*, add a <button> element. Use similar logic to *step 5* and *step 7* to change the content of the button, based on instance being set. The full button definition is like this:

```
<button type="submit" class="btn btn-primary">
  {% if instance %}Save{% else %}Create{% endif %}
</button>
```

Close {% block %} you opened in *step 7* with {% endblock %}:

```
</button>
{% endblock %}
```

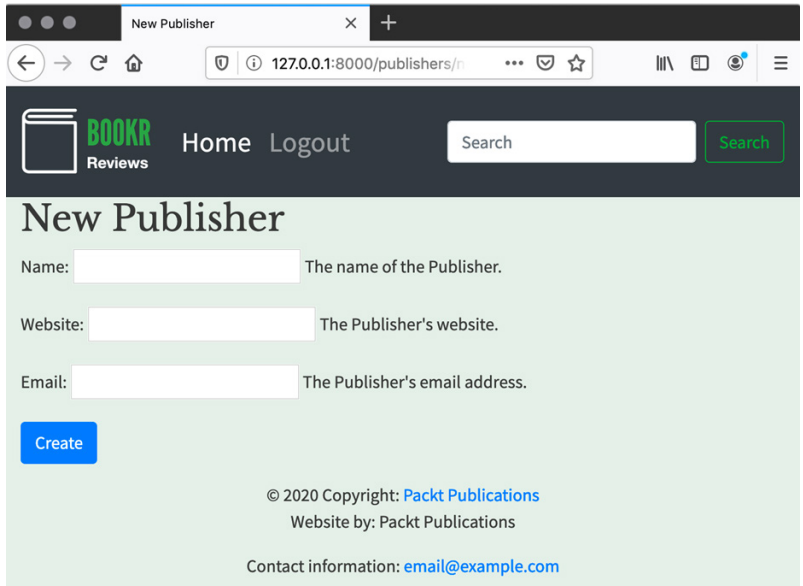
12. Open reviews/views.py. Update the second argument to the render call to "reviews/instance-form.html".

13. Update the context dictionary (the third argument to render). Add the instance key with the publisher value (the Publisher instance that was fetched from the database, or None for a creation). Also, add the model_type key, set to the Publisher string. You should retain the form key that was already there. The method key can be removed. After completing the previous steps and this one, your render call should look like this:

```
render(request, "reviews/instance-form.html",
  {"form": form, "instance": publisher,
    "model_type": "Publisher"})
```

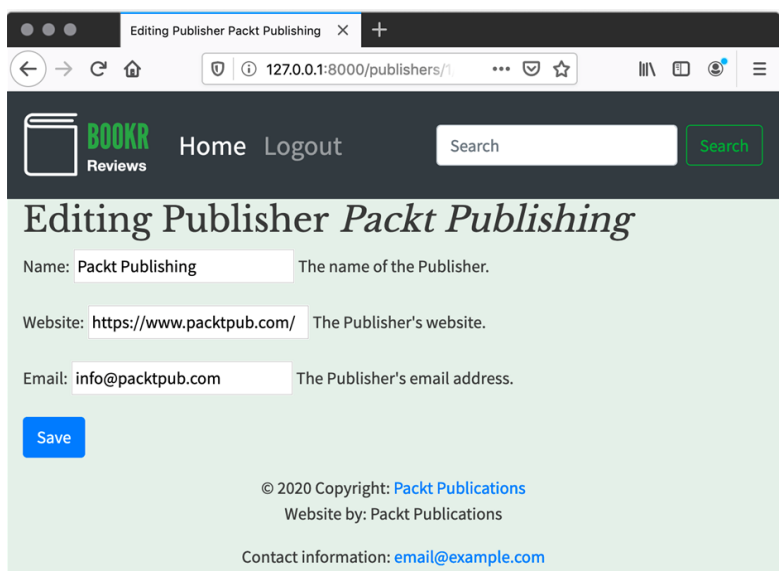
14. Delete the reviews/templates/form-example.html file.

After completing the activity, you should be able to create and edit Publisher instances. Create a new Publisher by visiting http://127.0.0.1:8000/publishers/new/. The page should look like *Activity Figure 7.2*:



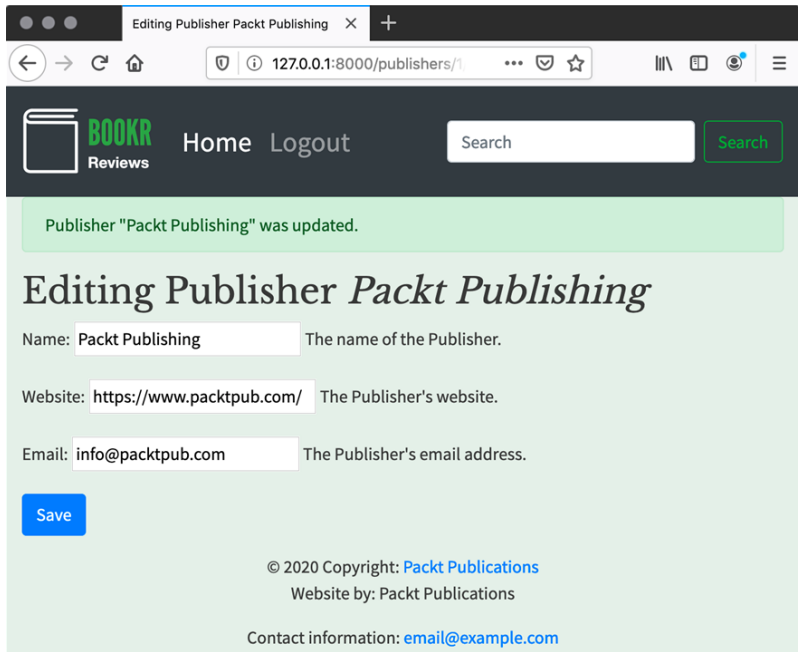
Activity Figure 7.2: The Publisher creation page

Once you have one or more publishers, you can visit `http://127.0.0.1:8000/publishers/<id>/` – for example, `http://127.0.0.1:8000/publishers/1/`. Your page should look like *Activity Figure 7.3*:



Activity Figure 7.3: The Publisher edit page

After saving `Publisher`, whether you were creating or editing, the success message should appear in Bootstrap style. This is shown in *Activity Figure 7.4*:



Activity Figure 7.4: A success message rendered as a Bootstrap alert

Activity 7.02 – reviewing the creation UI

The following steps will help you complete this activity:

1. Open `forms.py` inside the `reviews` app. At the top of the file, make sure you are importing the `Review` model. You will already be importing `Publisher`, like this:

```
from .models import Publisher
```

Import `Review` as well on this line:

```
from .models import Publisher, Review
```

Create a `ReviewForm` class that inherits from `forms.ModelForm`. Add a `class Meta` attribute to set the model to `Review`:

```
class ReviewForm(forms.ModelForm):
    class Meta:
        model = Review
```

Use the `exclude` attribute on the `ReviewForm` `Meta` attribute to exclude the `date_edited` and `book` fields so that they do not display on the form. `ReviewForm` should now look like this:

```
class ReviewForm(forms.ModelForm):
    class Meta:
        model = Review
        exclude = ["date_edited", "book"]
```

Note that `exclude` could be a tuple instead – that is, `("date_edited", "book")`.

Add a `rating` field – this will override the validation for the model's `rating` field. It should be of the `models.IntegerField` class and instantiated with `min_value=0` and `max_value=5`:

```
class ReviewForm(forms.ModelForm):
    class Meta:
        # Code truncated

    rating = forms.IntegerField(min_value=0,
    max_value=5)
```

The complete `forms.py` can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter07/Activity7.02/bookr/reviews/forms.py`.

2. Open `views.py` inside the `reviews` app. First, make sure to import the `Review` model. Consider the following line:

```
from .models import Book, Contributor, Publisher
```

Change it to the following:

```
from .models import Book, Contributor, Publisher, Review
```

Then, create a new view function called `review_edit`. It should take three arguments – `request` (required), `book_pk` (required), and `review_pk` (optional, and it should default to `None`):

```
def review_edit(request, book_pk, review_pk=None):
```

Fetch the `Book` instance with the `get_object_or_404` function, passing in `pk=book_pk` as a kwarg. Store it in a variable named `book`:

```
def review_edit(request, book_pk, review_pk=None):
    book = get_object_or_404(Book, pk=book_pk)
```

Fetch `Review`, which will be edited only if `review_pk` is not `None`. Use the `get_object_or_404` function again, passing in the `book_id=book_pk` and `pk=review_pk` kwargs. Store the return value in a variable named `review`. If `review_pk` is `None`, then just set `review` to `None`:

```
def review_edit(request, book_pk, review_pk=None):
# Code truncated
    if review_pk is not None:
        review = get_object_or_404
            (Review, book_id=book_pk, pk=review_pk)
    else:
    review = None
```

Note

Note that since `review_pk` is unique, we could fetch the review by just using `review_pk`. The reason that we use `book_pk` as well is to enforce the URLs so that a user cannot try to edit a review for a book that it does not belong to. It could get confusing if you could edit any review in the context of any book.

3. Make sure you have imported `ReviewForm` near the start of the file. You will already have a line importing `PublisherForm`:

```
from .forms import PublisherForm, SearchForm
```

4. Open `forms.py` inside the `reviews` app. At the top of the file, make sure you are importing the `Review` model. You will already be importing `Publisher`, like this:

```
from .models import Publisher
```

Import `Review` as well on this line:

```
from .models import Publisher, Review
```

Create a `ReviewForm` class that inherits from `forms.ModelForm`. Add a `class Meta` attribute to set the model to `Review`:

```
class ReviewForm(forms.ModelForm):
    class Meta:
        model = Review
```

Use the `exclude` attribute on the `ReviewForm` `Meta` attribute to exclude the `date_edited` and `book` fields so that they do not display on the form. `ReviewForm` should now look like this:

```
class ReviewForm(forms.ModelForm):
    class Meta:
        model = Review
        exclude = ["date_edited", "book"]
```

Note that `exclude` could be a tuple instead – that is, `("date_edited", "book")`.

Add a `rating` field – this will override the validation for the model's `rating` field. It should be of the `models.IntegerField` class and instantiated with `min_value=0` and `max_value=5`:

```
class ReviewForm(forms.ModelForm):
    class Meta:
        # Code truncated

    rating = forms.IntegerField(min_value=0,
    max_value=5)
```

The complete `forms.py` can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter07/Activity7.02/bookr/reviews/forms.py`.

5. Open `views.py` inside the `reviews` app. First, make sure to import the `Review` model. Consider the following line:

```
from .models import Book, Contributor, Publisher
```

Change it to the following:

```
from .models import Book, Contributor, Publisher,
Review
```

Then, create a new view function called `review_edit`. It should take three arguments – `request` (required), `book_pk` (required), and `review_pk` (optional, and it should default to `None`):

```
def review_edit(request, book_pk, review_pk=None):
```

Fetch the `Book` instance with the `get_object_or_404` function, passing in `pk=book_pk` as a kwarg. Store it in a variable named `book`:

```
def review_edit(request, book_pk, review_pk=None):
  book = get_object_or_404(Book, pk=book_pk)
```

Fetch `Review`, which is edited only if `review_pk` is not `None`. Use the `get_object_or_404` function again, passing in the `book_id=book_pk` and `pk=review_pk` kwargs. Store the return value in a variable named `review`. If `review_pk` is `None`, then just set `review` to `None`:

```
def review_edit(request, book_pk, review_pk=None):
  # Code truncated
  if review_pk is not None:
    review = get_object_or_404(Review,
        book_id=book_pk, pk=review_pk)
  else:
    review = None
```

Note

Note that since `review_pk` is unique, we could fetch the review by just using `review_pk`. The reason that we use `book_pk` as well is to enforce the URLs so that a user cannot try to edit a review for a book that it does not belong to. It could get confusing if you could edit any review in the context of any book.

6. Make sure you have imported `ReviewForm` near the start of the file. You will already have a line importing `PublisherForm`:

```
from .forms import PublisherForm, SearchForm
```

Just update it to add the `ReviewForm` import:

```
from .forms import PublisherForm, SearchForm, ReviewForm
```

In the `review_edit` function, check whether `request.method` is equal to the `POST` string. If so, instantiate the form and pass in `request.POST` and the `Review` instance (this was stored in the `review` variable, and might be `None`):

```
def review_edit(request, book_pk, review_pk=None):
  # Code from Steps 5-6 truncated
  if request.method == "POST":
    form = ReviewForm(request.POST, instance=review)
```

Check the validity of the form using the `is_valid()` method. If it is valid, then save the form using the `save()` method. Pass `False` to `save()` (this is the `commit` argument) so that `Review` is not yet committed to the database. We do this because we need to set the `book` attribute of `Review`. Remember the `book` value is not set in the form, so we set it to `Book`, whose context we are in.

The `save()` method returns the new or updated `Review`, and we store it in a variable called `updated_review`:

```
def review_edit(request, book_pk, review_pk=None):
  # Code truncated
  if request.method == "POST":
    # Code truncated
    if form.is_valid():
      updated_review = form.save(False)
      updated_review.book = book
```

You can check whether we are creating or editing a `Review` based on the value of the `review` variable. This is the `Review` you fetched from the database – or was set to `None`. If it is `None`, then you must be creating `Review` (as the view could not load one to edit). Otherwise, you are editing `Review`. So, if `review` is not `None`, then set the `date_edited` attribute of `updated_review` to the current date and time. Make sure you import the `timezone` module from `django.utils` (you should put this line near the start of the file):

```
from django.utils import timezone
```

Then, implement the edit/creation check (it is an edit if `review` is not `None`), and set the `date_edited` attribute:

```
def review_edit(request, book_pk, review_pk=None):
  # Code truncated
  if request.method == "POST":
    form = ReviewForm(request.POST, instance=review)
    if form.is_valid():
      # Code truncated
      if review is None:
        pass # This branch is filled in Step 4
      else:
        updated_review.date_edited = timezone.now()
```

7. Add a call to `updated_review.save()`, which should happen regardless of an edit or create. Create it after the `if/else` branch you added in *step 3*:

```
def review_edit(request, book_pk, review_pk=None):
  # Code from Steps 5-7 truncated
  if request.method == "POST":
    form = ReviewForm(request.POST, instance=review)
    if form.is_valid():
      # Code truncated
      if review is None:
        pass # This branch is filled next
      else:
        updated_review.date_edited = timezone.now()

      updated_review.save()
```

8. Now, register the success messages inside the `if/else` branch created in *step 3*. Use the `messages.success` function, and the text should be either `Review for "<book>" created` or `Review for "<book>" updated`. Regardless of an edit or create, you should return a redirect HTTP response back to the `book_detail` URL using the `redirect` function. You'll also need to pass `book.pk` to this function, as it is required to generate the URL:

```
def review_edit(request, book_pk, review_pk=None):
  # Code truncated
  if request.method == "POST":
    form = ReviewForm(request.POST, instance=review)
    if form.is_valid():
      # Code truncated
    if review is None:
      messages.success
      (request, "Review for "{}" created."
```

```
            .format(book))
    else:
      updated_review.date_edited = timezone.now()
      messages.success(request, "Review for "{}"
          updated.".format(book))

    return redirect("book_detail", book.pk)
```

9. Now, create the non-`POST` branch. This simply instantiates `ReviewForm` and passes in the `Review` instance (again, this might be `None`):

```
def review_edit(request, book_pk, review_pk=None):
  # Code truncated

  if request.method == "POST":
    # Code truncated
  else:
    form = ReviewForm(instance=review)
```

10. The last line of the function is to call the `render` function and return the result. This code will be called if the method is not `POST` or the form is not valid.

    The arguments to `render` are `request`, the `instance-form.html` template (`"reviews/instance-form.html"`), and the context dictionary. The context dictionary should contain these keys and values:

    - `form`: The form variable
    - `instance`: The `Review` instance (the `review` variable)
    - `model_type`: The `Review` string
    - `related_instance`: The `Book` instance (the `book` variable)
    - `related_model_type`: The `Book` string

    The `render` call should look like this:

```
def review_edit(request, book_pk, review_pk=None):
  # Code truncated

  return render(request, "reviews/instance-form.html",
      {"form": form, "instance": review,
      "model_type": "Review",
      "related_instance": book,
      "related_model_type": "Book"})
```

The complete `views.py` can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter07/Activity7.02/bookr/reviews/views.py`.

11. Open `instance-form.html` in the `reviews` app's templates directory. Locate the closing `</h2>` tag, and underneath, add an `if` template tag. It should check that both `related_instance` and `related_model_type` are set. Its contents should be a `<p>` element displaying `For {{ related_model_type }} <em>{{ related_instance }}</em>`. The code you add should look like this:

```
{% if related_instance and related_model_type %}
  <p>For {{ related_model_type }} <em>
    {{ related_instance }}</em></p>
{% endif %}
```

The complete `instance-form.html` can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter07/Activity7.02/bookr/reviews/templates/reviews/instance-form.html`.

12. Open the `reviews` app's `urls.py` file. Add these two mappings to `urlpatterns`:

```
path('books/<int:book_pk>/reviews/new/',
    views.review_edit, name='review_create'),
path('books/<int:book_pk>/reviews/<int:review_pk>/',
    views.review_edit, name='review_edit'),
```

Note that the order of URL patterns does not matter in this case, so you may have put the new maps in a different place in the list. This is fine. The complete `urls.py` file can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter07/Activity7.02/bookr/reviews/urls.py`.

13. Open `book_detail.html`. Locate the `{% endblock %}` closing template tag for the `content` block. This should be the last line of the file. On the line before this, add a link using an `<a>` element. Its `href` attribute should be set using the `url` template tag, with the name of the URL as defined in the map – `{% url 'review_create' book.pk %}`. The classes on `<a>` should be `btn` and `btn-primary`. The full link code is as follows:

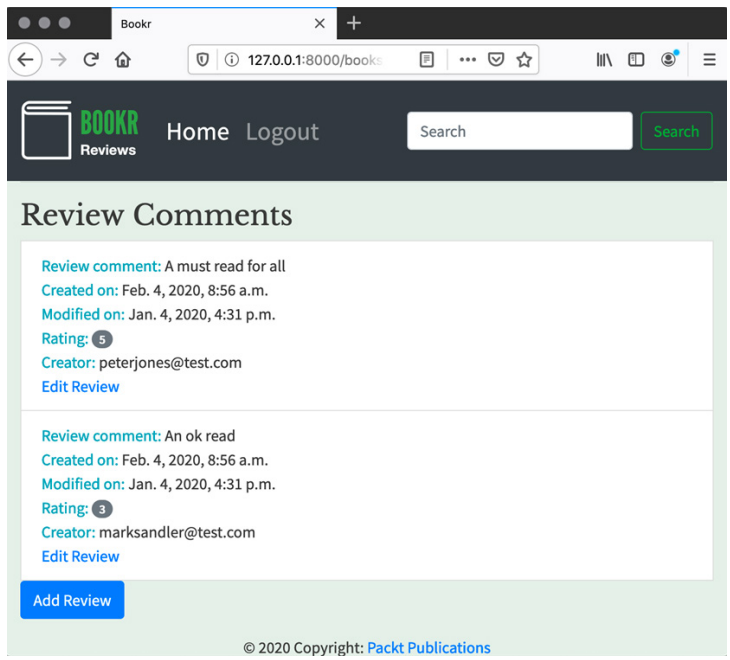
```
<a class="btn btn-primary" href="{% url
  'review_create' book.pk %}">Add Review</a>
```

14. Locate the `reviews` iterator template tag, `{% for review in reviews %}`, and then its corresponding `{% endfor %}` closing template tag. Before this closing template tag is a closing `</li>`, which closes the list item that contains all the `Book` data. You should add the new `<a>` element before the closing `</li>`. Generate the `href` attribute content using the `url` template tag and the name of the URL defined in the map – `{% url 'review_edit' book.pk review.pk %}`. The full link code is as follows:

```
<a href="{% url 'review_edit' book.pk review.pk %}">
  Edit Review</a>
```

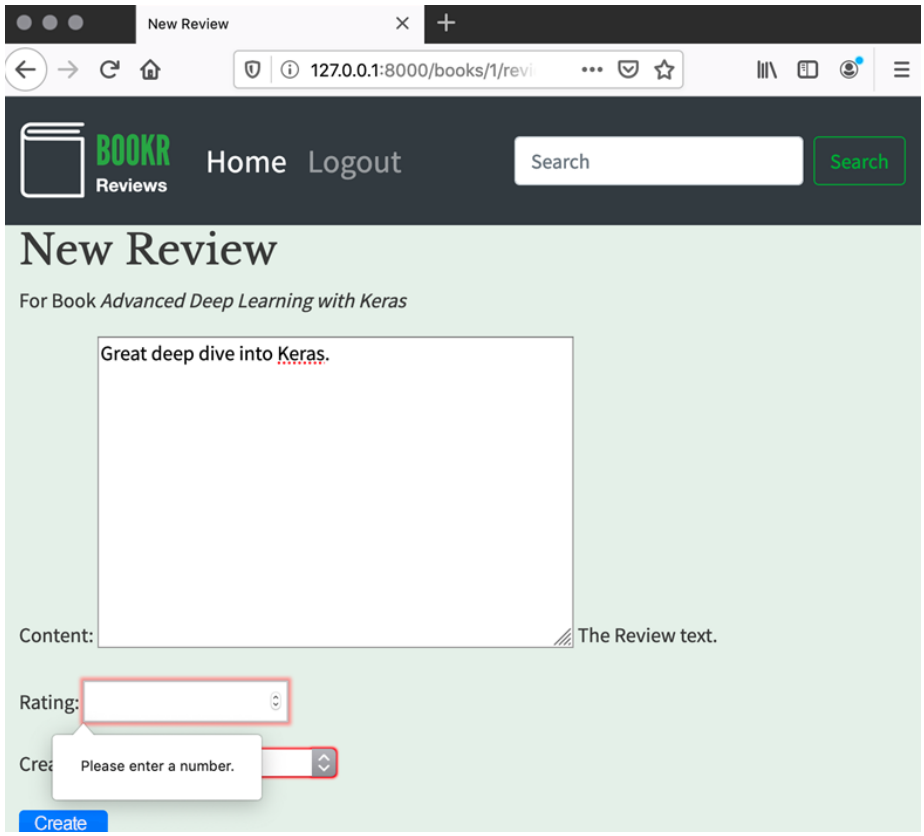
The complete `book_detail.html` file can be found at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter07/Activity7.02/bookr/reviews/templates/reviews/book_detail.html`.

Start the Django development server if it is not already running. The first URL assumes you already have at least one book in your database. Visit `http://127.0.0.1:8000/books/1/`. You should see your new **Add Review** button:



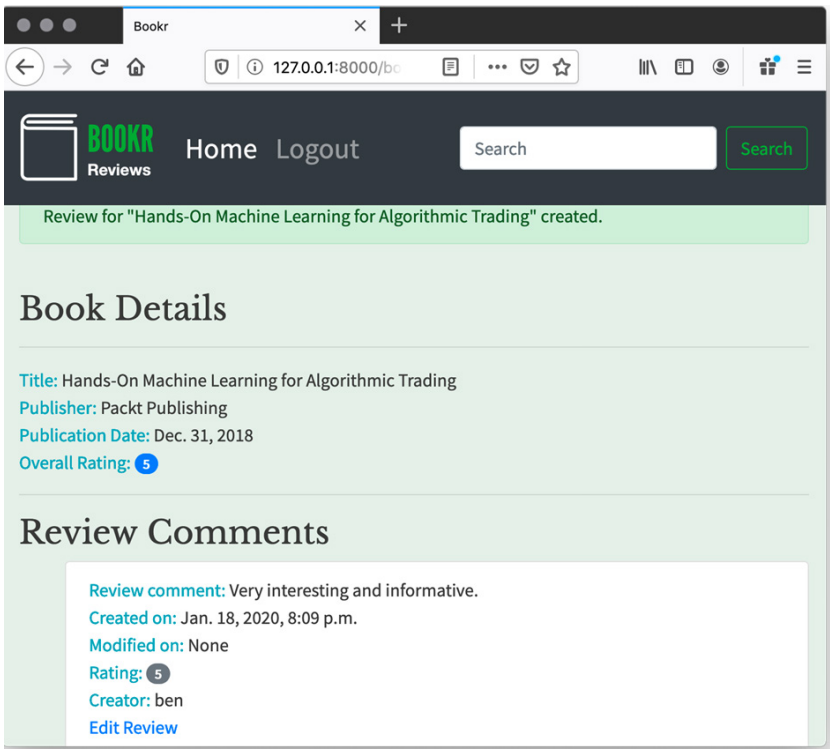
Activity Figure 7.5: The book details page with an Add Review button

Clicking it will take you to the review creation view for the book. If you try submitting the form with missing fields, your browser should use its validation rules to prevent submission (*Activity Figure 7.6*):



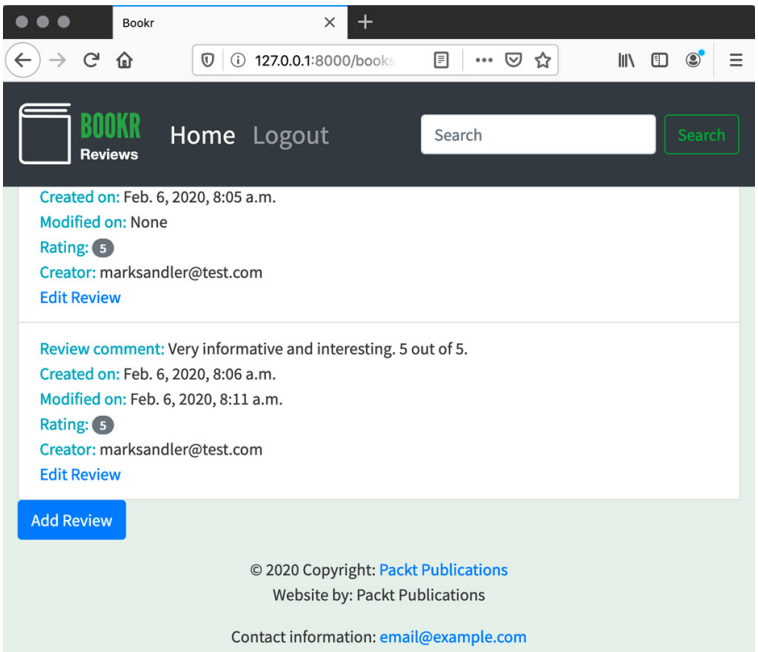
Activity Figure 7.6: The review creation form with a missing rating

After creating the form, you are taken back to the **Book Details** page, and you can see the review you added, along with a link to go back and edit the `Review`:



Activity Figure 7.7: A new review added, with an Edit Review link

Click the **Edit Review** link and make some changes to the form. Save it, and you will be redirected back to the **Book Details** view again – this time, the **Modified on** field should show the current date and time (see *Activity Figure 7.8*):



Activity Figure 7.8: The Modified on date populated after editing a review

In this activity, we used `ModelForm` to add a page where users can save a review. Some custom validation rules on `ModelForm` ensured that the review's rating was between `0` and `5`. We also created a generic instance form template that can render different types of `ModelForm`.

Chapter 8, Media Serving and File Uploads

Activity 8.01 – image and PDF uploads of books

The following steps will help you complete this activity:

1. Open the project's `settings.py` file. Down the bottom, add these two lines to set `MEDIA_ROOT` and `MEDIA_URL`:

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

The file can now be saved and closed. Your file should now look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/bookr/settings.py`.

2. Open `urls.py`. Above the other imports, add these two lines:

```
from django.conf import settings
from django.conf.urls.static import static
```

These will import the Django settings and the built-in `static` view, respectively.

Then, after your `urlpatterns` definition, conditionally add a map to the `static` view if `DEBUG` is `true`:

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

Save and close `urls.py`. It should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/bookr/urls.py`.

3. Open the `reviews` app's `models.py`. To the book model, add the `cover` field attribute with this code:

```
cover = models.ImageField(null=True, blank=True,
    upload_to="book_covers/")
```

This will add an `ImageField`, which is not required, and store uploads to the `book_covers` subdirectory of `MEDIA_ROOT`.

The `sample` field is added in a similar manner:

```
sample = models.FileField(null=True, blank=True,
    upload_to="book_samples/")
```

This field is also not required and allows uploads of any file type, storing them in the `book_samples` subdirectory of `MEDIA_ROOT`.

Save the file. It should look like this now: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/reviews/models.py`.

4. Open a terminal and navigate to the Bookr project directory. Run the `makemigrations` management command:

```
python3 manage.py makemigrations
```

This will generate the migration to add the `cover` and `sample` fields to `Book`.

You should see similar output to this:

```
(bookr)$ python3 manage.py makemigrations
Migrations for 'reviews':
  reviews/migrations/0006_auto_20200123_2145.py
    - Add field cover to book
    - Add field sample to book
```

Then, run the `migrate` command to apply the migration:

```
python3 manage.py migrate
```

You should see similar output to this:

```
(bookr)$ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
  reviews, sessions
Running migrations:
  Applying reviews.0006_auto_20200123_2145... OK
```

5. Back in PyCharm, open the `reviews` app's `forms.py`. You first need to import the `Book` model at the top of the file. Consider the following line:

```
from .models import Publisher, Review
```

Change this line to the following:

```
from .models import Publisher, Review, Book
```

Then, at the end of the file, create a `BookMediaForm` class as a subclass of `forms.ModelForm`. Using the `class Meta` attribute, set `model` to `Book`, and `fields` to `["cover", "sample"]`. Your completed `BookMediaForm` should look like this:

```
class BookMediaForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ["cover", "sample"]
```

You can save and close this file. The complete file should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/reviews/forms.py`.

6. Open the `reviews` app's `views.py`. Import the image manipulation libraries (refer to *step 4* of *Exercise 8.05 – image uploads using Django forms* to install Pillow, if you haven't already installed it in your virtual environment):

```
from io import BytesIO
from PIL import Image
from django.core.files.images import ImageFile
```

You'll also need to import the `BookMediaForm` class you just created. Find the following import line:

```
from .forms import PublisherForm, SearchForm, ReviewForm
```

Change it to this:

```
from .forms import PublisherForm, SearchForm, ReviewForm,
BookMediaForm
```

At the end of the file, create a view function called `book_media`, which accepts two arguments – `request` and `pk`:

```
def book_media(request, pk):
```

7. The view will follow a pattern similar to what we have done before. First, fetch the `Book` instance with the `get_object_or_404` shortcut:

```
def book_media(request, pk):
    book = get_object_or_404(Book, pk=pk)
```

If `request.method` is `POST`, then instantiate `BookMediaForm` with `request.POST`, `request.FILES`, and the `Book` instance:

```
    if request.method == "POST":
        form = BookMediaForm(request.POST,
        request.FILES, instance=book)
```

Check whether the form is valid, and if so, save the form. Make sure to pass `False` as the `commit` argument to `save`, since we want to update and resize the image before saving the data:

```
        if form.is_valid():
            book = form.save(False)
```

Create a reference to the uploaded cover. This is mostly just a shortcut so that you don't have to type `form.cleaned_data["cover"]` all the time:

```
            cover = form.cleaned_data.get("cover")
```

Next, resize the image. Use the `cover` variable as a reference to the uploaded file. You only want to perform operations on the image if it is not `None`:

```
            if cover:
```

This code is similar to that demonstrated in the *Writing PIL images to ImageField* section. You should refer to that section for a full explanation. Essentially, you are using PIL to open the uploaded image file, and then resize it so that its maximum dimension is 300 pixels. You then write the data to `BytesIO` and save it on the model, using a Django `ImageFile`:

```
                image = Image.open(cover)
                image.thumbnail((300, 300))
                image_data = BytesIO()
                image.save(fp=image_data,
                           format=cover.image.format)
                image_file = ImageFile(image_data)
                book.cover.save(cover.name,
                                image_file)
```

After doing the updates, save the `Book` instance. The form might have been submitted with the `clear` option set on the `cover` or `sample` fields, so this will clear those fields if so:

```
            book.save()
```

Then, we register the success message and redirect back to the `book_detail` view:

```
            messages.success(request, "Book "{}"
         was successfully updated.".format(book))
            return redirect("book_detail", book.pk)
```

This `else` branch is for the non-`POST` request case. Instantiate the form with just the `Book` instance:

```
    else:
        form = BookMediaForm(instance=book)
```

8. The last thing to do in `book_media` is to render `instance-form.html` and pass the instance (the `book` variable), the form (the `form` variable), `model_type` (the `Book` string), and `is_file_upload` (`True`) in the context dictionary:

```
return render(request, "reviews/instance-form.html",
    {"instance": book, "form": form,
    "model_type": "Book","is_file_upload":True})
```

You will use the `is_file_upload` flag in the next step. Once you have completed *steps 6–8*, your `book_media` function should look like `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/reviews/views.py`:

9. Open the `instance-form.html` file inside the `reviews` app's `templates` directory. Use the `if` template tag to add the `enctype="multipart/form-data"` attribute to the form, if `is_file_upload` is `True`:

```
<form method="post" {% if is_file_upload %}
  enctype="multipart/form-data"{% endif %}>
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit" class="btn btn-primary">
      {% if instance %}Save{% else %}Create{% endif %}
    </button>
</form>
```

Save and close the file. It should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/reviews/templates/reviews/instance-form.html`.

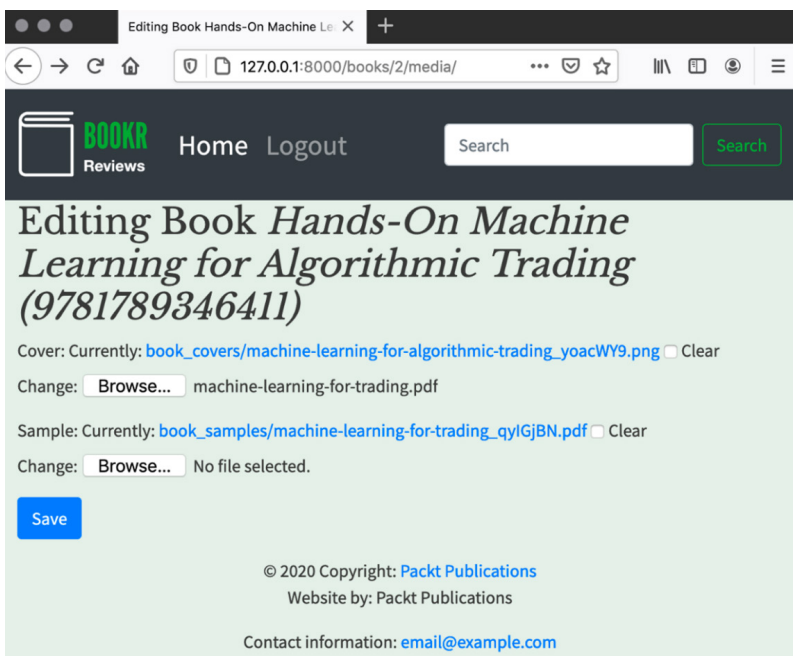
10. Open the `reviews` app's `urls.py`. In `urlpatterns`, add a mapping like this:

```
urlpatterns = [...
    path('books/<int:pk>/media/',
    views.book_media, name='book_media')
```

Your `urlpatterns` should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.01/bookr/reviews/urls.py`.

Remember that the order of your `urlpatterns` might be different.

You should now be able to start the Django development server if it's not already running; then, view the `Book Media` page in your browser – for example, at `http://127.0.0.1:8000/books/2/media/`. The following figure shows this:



Activity Figure 8.1: BookMediaForm with existing values

Try uploading some files and checking whether you can see them in the media directory. Check the sizes of the images that you upload too, and note that they have been resized.

Activity 8.02 – displaying cover and sample links

The following steps will help you complete this activity:

1. Open the `reviews` app's `book_detail.html` template. After the first `<hr>` tag, add an `if` template tag that checks for the presence of `book.cover`. Inside it, put your `<img>` tag. Its `src` should be set from `book.cover.url`. Add a `<br>` tag after the `<img>` tag. The code you add should look like this:

```
{% if book.cover %}
  <img src="{{ book.cover.url }}">
  <br>
{% endif %}
```

2. Perform a similar check for the presence of `book.sample`, and if it is set, display an `info` line similar to the existing ones. The `<a>` tag's `href` attribute should be set using `book.sample.url`. The code you add here should look like this:

```
{% if book.sample %}
  <span class="text-info">Sample: </span>
  <span><a href="{{ book.sample.url }}">
    Download</a></span>
  <br>
{% endif %}
```

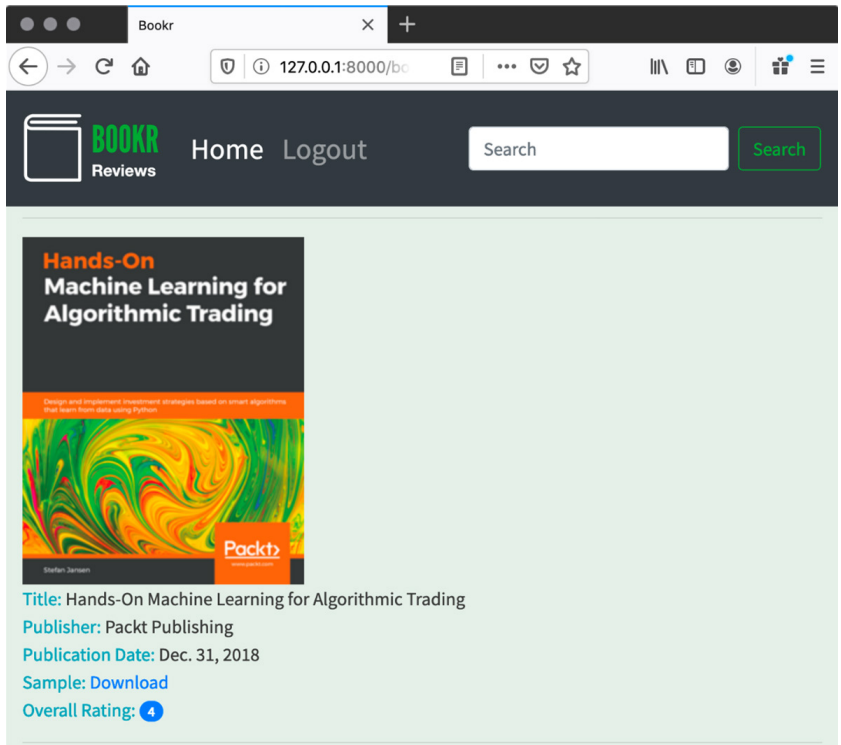
3. Scroll to the end of the file where there the `Add Review` link is located. Underneath it, add another `<a>` tag, and generate its `href` content using the `url` template tag. This should use `'book_media'` and `book.pk` as its arguments. Make sure you include the same classes on the `<a>` tag as the existing `Add Review` link so that the link displays as a button.

   This is the code you should add:

```
<a class="btn btn-primary" href="
  {% url 'book_media' book.pk %}">Media</a>
```

   Once again, here is the code that was added in context with the existing code. The complete file should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter08/Activity8.02/bookr/reviews/templates/reviews/book_detail.html`.

Now, you can start the Django development server if it is not already running. You can view a **Book Details** page (for example, `http://127.0.0.1:8000/books/1/`) and then click on the new `Media` link to get to the media page for `Book`. After uploading a cover image or sample file, you will be taken back to the **Book Details** page, where you will see the cover and a link to the sample file (*Activity Figure 8.2*):



Activity Figure 8.2: The book cover and sample link displayed

In this activity, we added the display of a book's cover image and a link to its sample to the **Book Details** page, as well as a link from the **Book Details** page to the **Book Media** page. This enhances the look of Bookr by showing an image and gives users a more in-depth look into the `Book` that was reviewed.

Chapter 9, Sessions and Authentication

Activity 9.01 – authentication-based content using conditional blocks in templates

The following steps will help you complete this activity:

1. On the `book_detail` template, as found in the `reviews/templates/reviews/book_detail.html` file, hide the `Add Review` and `Media` buttons from non-authenticated users. Wrap the two links in an `if user.is_authenticated … endif` block:

```
{% if user.is_authenticated %}
  <a class="btn btn-primary" href="{% url
  'review_create' book.pk %}">Add Review</a>
```

```
    <a class="btn btn-primary" href="{% url 'book_media'
      book.pk %}">Media</a>
  {% endif %}
```

2. In that same file, wrap the header in an `if user.is_authenticated ... endif` block:

```
{% if user.is_authenticated %}
  <h3>Be the first one to write a review.</h3>
{% endif %}
```

3. In the same template, make the `Edit Review` link only appear for staff or the user that wrote the review. The conditional logic for the template block is very similar to the conditional logic that we used in the `review_edit` view in *Exercise 9.03 – Adding authentication decorators to the views*. We can make use of the `is_staff` property of the user. We need to compare the user's `id` with that of the review's creator to determine that they are the same user:

```
{% if user.is_staff or user.id == review.creator.id %}
  <a href="{% url 'review_edit' book.pk review.pk %}">
    Edit Review</a>
{% endif %}
```

4. From the `templates` directory in the main `bookr` project, modify `base.html` so that it displays the currently authenticated user's username to the right of the search form in the header, linking to the user profile page. Again, we can use the `user.is_authenticated` attribute to determine whether the user is logged in and the `user.username` attribute for the username. This conditional block follows the search form in the header:

```
{% if user.is_authenticated %}
  <a class="nav-link" href="/accounts/profile">User:
    {{ user.username }}</a>
{% endif %}
```

By tailoring content to a user's authentication status and permissions, we create a smoother and more intuitive user experience. The next step is to store user-specific information in sessions so that the project can incorporate the user's preferences and interaction history.

Activity 9.02 – using session storage for the book search page

The following steps will help you complete this activity:

1. Edit the `book_search` view to retrieve `search_history` from the session in the `reviews/views.py` file:

```
def book_search(request):
    search_text = request.GET.get("search", "")
```

```
    search_history =
        request.session.get('search_history', [])
```

2. If the form has received valid input and a user is authenticated, append the search option, `search_in`, and search text, `search`, to the session's `search_history` list:

```
    if form.is_valid() and
    form.cleaned_data["search"]:
        search = form.cleaned_data["search"]
        search_in = form.cleaned_data.get("search_in")
            or "title
        if search_in == "title":
            …
        if request.user.is_authenticated:
            search_history.append([search_in, search])
            request.session['search_history'] =
                search_history
```

3. If the form hasn't been filled in (for example, when the page is first visited), render the form with the previously used search option selected, either `Title` or `Contributor`:

```
    elif search_history:
        initial = dict(search=search_text,
            search_in=search_history[-1][0])
        form = SearchForm(initial=initial)
```

The complete `book_search` function should look like this:

bookr/reviews/views.py

```
def book_search(request):
    search_text = request.GET.get("search", "")
    search_history =
        request.session.get('search_history', [])

    form = SearchForm(request.GET)

    books = set()

    if form.is_valid() and
    form.cleaned_data["search"]:
```

You can view the complete code at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter09/Activity9.02/bookr/reviews/views.py`.

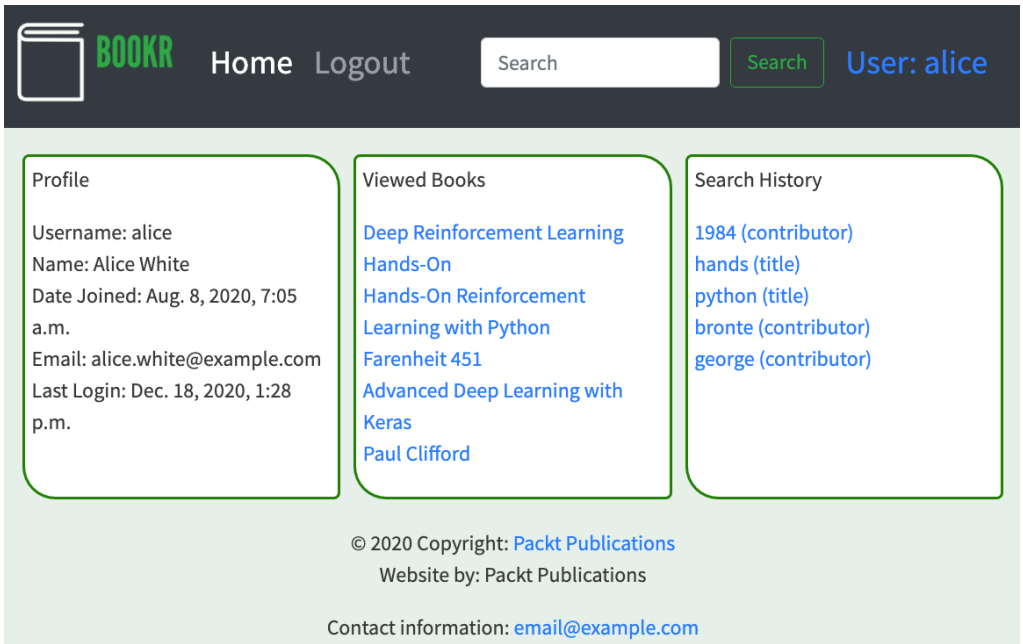
4. In the profile template, `templates/profile.html`, include an additional `infocell` division for `Search History`, as follows:

```
<style>
…

</style>

  <div class="flexrow" >
    <div class="infocell" >
      <p>Profile</p>
      …
    </div>

    <div class="infocell" >
      <p>Viewed Books</p>
      …
    </div>

    <div class="infocell" >
      <p>Search History</p>
      …
    </div>

  </div>
```

5. List the search history as a series of links to the book search page. The complete `Search History` division will look like this:

```
  <div class="infocell" >
    <p>Search History</p>
    <p>
      {% for search_in, search in
        request.session.search_history %}
      <a href="{% url 'book_search' %}?search=
        {{search|urlencode}}&search_in={{ search_in}}"
          >{{ search }} ({{ search_in }})</a>
      <br>
      {% empty %}
        No search history found.
      {% endfor %}
    </p>
  </div>
```

Once you have completed these changes and made some searches, the profile page will look something like this:



Activity Figure 9.1: The profile page with the Search History infocell

If it is not already running, you will need to start the Django server using this command:

```
python manage.py runserver
```

To view the profile page, you will need to log in and click the `User` link in the top-right corner of the web page.

Keeping form preferences in session data is a useful technique to improve the user experience. We naturally expect settings to still be the way that we last saw them, and it is frustrating to find all the values cleared in a complicated form that we use repeatedly.

Chapter 10, Advanced Django Admin and Customizations

Activity 10.01 – building a custom admin dashboard with built-in search

1. As a first step, create an admin site application in the bookr project (if not created already), which you will use to build your custom admin application. To create this app, run the following command from the base directory of your project:

```
python manage.py startapp bookr_admin
```

2. With the admin site application now created, open the `admin.py` file under the `bookr_admin` directory and replace its contents with the following code:

```
from django.contrib import admin

class BookrAdmin(admin.AdminSite):
  site_header = "Bookr Administration Portal"
  site_title = "Bookr Administration Portal"
  index_title = "Bookr Administration"
```

In the preceding code sample, you first created a class named `BookrAdmin`, which inherits from the `AdminSite` class provided by Django's admin module. You have also customized the site properties to make the admin panel display the text you want. Your `admin.py` file should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/tree/main/Chapter10/Activity10.01/bookr_admin/admin.py`.

3. Now, with the class created, the next step involves making sure that your application will be recognized as a default admin site application. You should have already done this in *Exercise 10.02 – overriding the default admin site*. Consequently, your `apps.py` file under the `bookr_admin` directory should look like this:

```
from django.contrib.admin.apps import AdminConfig


class BookrAdminConfig(AdminConfig):
    default_site = 'bookr_admin.admin.BookrAdmin'
```

If not, replace the contents of the file with the code provided in the preceding code block.

The `apps.py` file should look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/tree/main/Chapter10/Activity10.01/bookr_admin/apps.py`.

4. The next step involves making sure that Django uses `bookr_admin` as the administration application. To check this, open the `settings.py` file under `bookr` and validate whether the following line is present in the `INSTALLED_APPS` section. If not, add it at the top of the section:

```
'bookr_admin.apps.BookrAdminConfig'
```

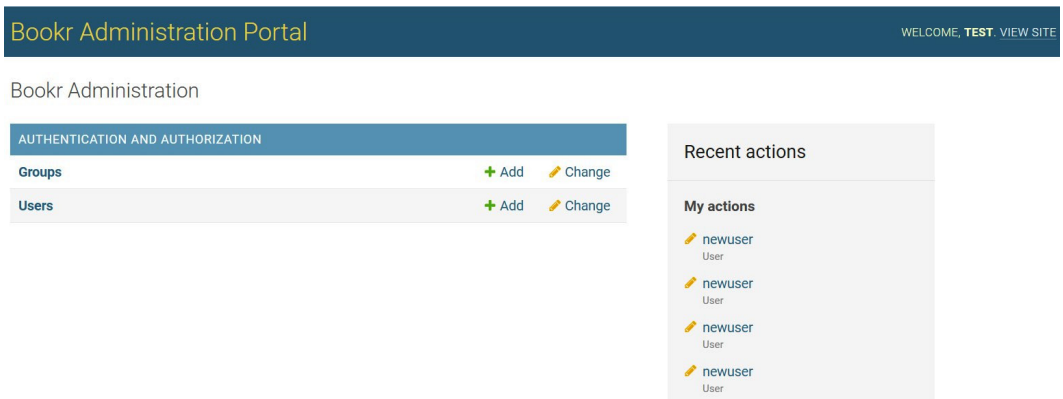
Your `INSTALLED_APPS` section should resemble the one shown here:

```
INSTALLED_APPS = [
    'bookr_admin.apps.BookrAdminConfig',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'reviews'
]
```

5. Once the `settings.py` file is configured to use `bookr_admin`, the default admin site should now be replaced by the instance you provided. To validate whether the changes you made worked, run the following command:

```
python manage.py runserver localhost:8000
```

Then, navigate to `http://localhost:8000/admin`. You will see a page that resembles the one shown here:



Activity Figure 10.1: The Bookr Administration Portal landing page

> **Note**
>
> The preceding screen will vary, depending on the version of the `admin.py` file (in the reviews app) you are using. If you're continuing from *Chapter 9*, *Sessions and Authentication*, the Book model should be visible. If it is, you can skip *step 6*. Still, it is recommended that you use that step to cross-validate your code.

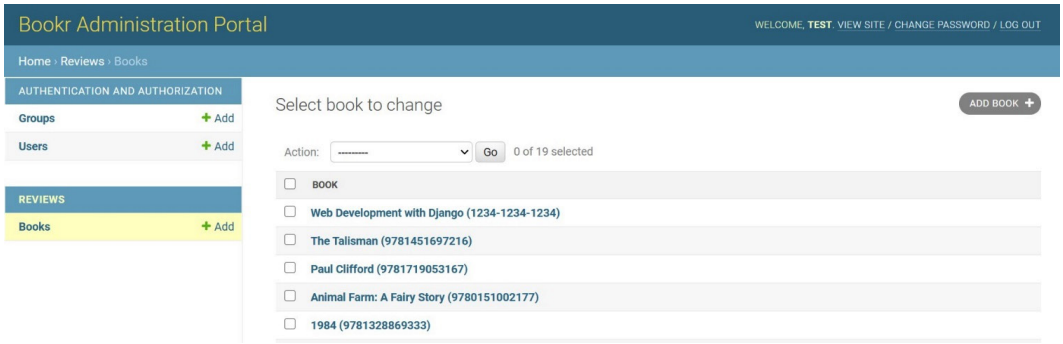
You now have a custom admin site up and running.

6. If there are no models registered to your admin site, the list of books still won't show up. To now view the list of books in the admin panel, open the `admin.py` file under the `reviews` app directory and validate whether the highlighted code blocks are present therein. If not, add them:

```
from django.contrib import admin

from reviews.models import (Publisher, Contributor, Book,
BookContributor, Review)

# Register your models here.
admin.site.register(Publisher)
admin.site.register(Contributor, ContributorAdmin)
admin.site.register(Book)
```

In the preceding code sample, you imported the `Book` model from our reviews app and registered it with the admin site. The `Book` model contains the records of all of the books that you have registered in Bookr, along with details about their publishers and publication dates.

Following this change, if you reload the admin site, you will see that the `Book` model starts to show up in the admin panel, and clicking it should show you something similar to the following screenshot:



Activity Figure 10.2: The Bookr Administration Portal Books model view

7. Now, to make sure that Bookr admins can search books by their name or the publisher's name on the admin site, you need to create a `ModelAdmin` interface for the `Book` model such that the behavior of the `Book` model can be customized inside the admin site. To do this, reopen the `admin.py` file under the `reviews` app directory and add the following class to the file:

```
class BookAdmin(admin.ModelAdmin):
   model = Book
    list_display = ('title', 'isbn', 'get_publisher',
        'publication_date')
    search_fields = ['title', 'publisher__name']

    def get_publisher(self, obj):
        return obj.publisher.name
```

The preceding class defines the `ModelAdmin` interface for our Book model. Inside this class, we define a couple of properties.

You begin by mentioning the model to which this `ModelAdmin` interface applies, by specifying the `model` property to point to our `Book` model:

```
model = Book
```

The next thing to do is to select the fields that are to be shown in the admin site table when someone clicks the `Book` model. This is done by setting the `list_display` property to the set of fields to display:

```
list_display = ('title', 'isbn', 'get_publisher',
'publication_date')
```

As you can see, you selected the `title`, `isbn`, and `publication_date` fields from the Book model. But what is this `get_publisher` field?

The `list_display` property can take input that includes the list of attributes of the Model class, as well as the name of any callable, and print the value returned by the callable. In this case, `get_publisher` is a callable defined inside the `BookAdmin` class.

The next thing you do is add a `search_fields` property and point it to use the title and `publisher__name` fields. This adds the capability in the admin site to search for books, either by their title or the name of the publisher.

The last thing you do in the class involves defining our `get_publisher()` callable. This callable is responsible for returning the name of the publisher from a book record. This callable is required because `Publisher` is mapped as a foreign key inside the `Book` model, and hence, you need to retrieve the name field of the publisher using the object reference you obtained.

The callable takes a single parameter, `obj`, which refers to the current object being parsed (the method is called for every object that we iterate upon), and returns the `name` field from the `publisher` object inside our result:

```
def get_publisher(self, obj):
    return obj.publisher.name
```

8. Once the preceding step is complete, you need to make sure that `ModelAdmin` is assigned to the Book model for use in our admin site. To do this, edit the `admin.site.register` call inside the `admin.py` file under the reviews app to make it look like the one shown here:

```
admin.site.register(Book, BookAdmin)
```

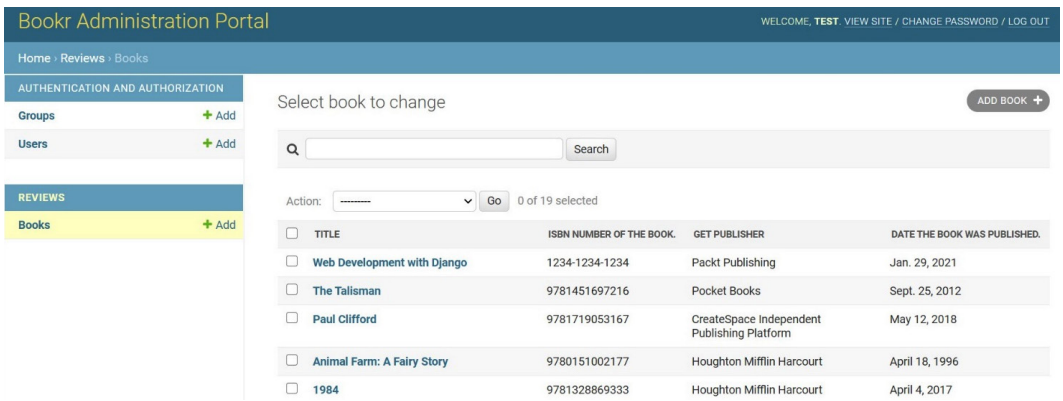
The `admin.py` file under the `reviews` app should now look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter10/Activity10.01/reviews/admin.py`.

9. Now that we are done with making changes, let's run the application by running the following command:

```
python manage.py runserver localhost:8000
```

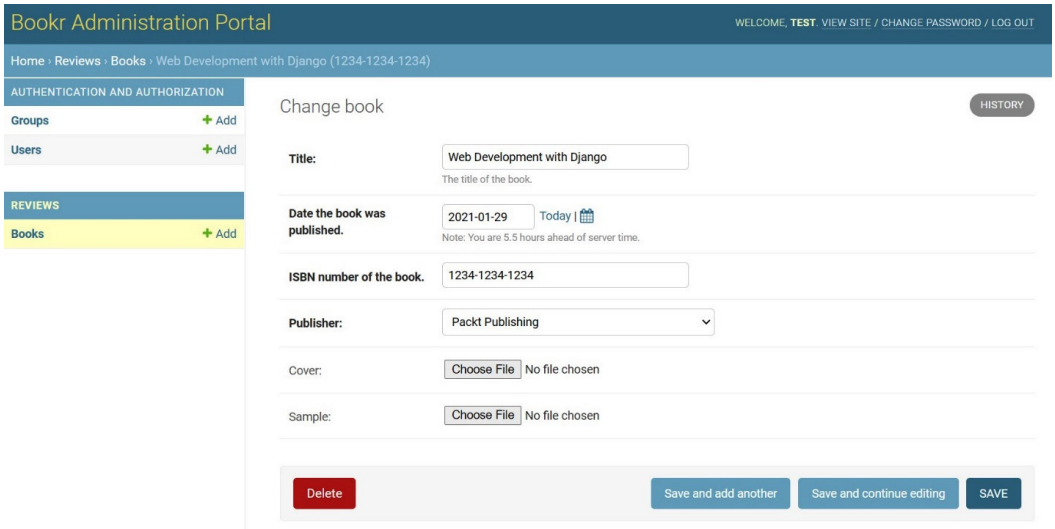
Then, navigate to `http://localhost:8000/admin`.

Once you are on the page, you can click on the Book model being shown, and it should redirect you to a page that resembles the following screenshot:



Activity Figure 10.3: Book model customizations in the Bookr Administration Portal view

Note the difference between our earlier page and the page we just loaded; we can see that the page now lists the fields that we selected in our `ModelAdmin` class definition, and also provides us with an option to search the books. We can insert either the name of a book or its publisher in the search box. Upon selecting a book, we have the option to either modify or delete it:



Activity Figure 10.4: Deleting a book record using the Administration Portal

If you are getting results that resemble those in *Activity Figure 10.4*, you have successfully completed this activity.

Chapter 11, Advanced Templating and Class-Based Views

Activity 11.01 – rendering details on the user profile page using inclusion tags

The following steps will help you complete this activity:

1. For this activity, you are going to reuse a lot of work that you have done in the previous chapters regarding building the `Bookr` app. Instead of recreating things from scratch, let's focus on those areas that are unique to this activity.

   Now, to begin introducing a custom inclusion tag, which will be used to render the list of books read by our user in the user profile, we first create a directory named `templatetags` under the `reviews` app directory, and then create a new file named `profile_tags.py` inside it.

2. With the file creation complete, the next step is to build the template tag. To do this, open the `profile_tags.py` file that you created in *step 1* and add the following code to it:

```
from django import template
from reviews.models import Review

register = template.Library()

@register.inclusion_tag('book_list.html')
def book_list(username):
    """Render the list of books read by a user.

    :param: str username The username for whom the
    books should be fetched

    :return: dict of books read by user
    """
    reviews = Review.objects.filter
        (creator__username__contains=username)
    book_list = [review.book.title for review in
        reviews]
    return {'books_read': book_list}
```

In this code, you did a few interesting things. Let's walk through them step by step.

You first imported Django's `template` module, which will be used to set up the template tags library:

```
from django import template
```

After that, you imported the model, which is used to store all the reviews by the user. This is done based on the assumption that a review is written by a user only for books that they have read, and hence, you used this `Review` model to derive the books read by a user:

```
from reviews.models import Review
```

Next, you initialized an instance of the `template` library. This instance will be used to register your custom template tags with Django's templating system:

```
register = template.Library()
```

Finally, you created the custom template tag named `book_list`. This tag will be an inclusion tag because, based on the data it generates, it renders its own template to show the results of the query for the books read based on the username of the user.

To render the results, you used the template code located inside the `book_list.html` template file:

```
@register.inclusion_tag('book_list.html')
def book_list(username):
```

Inside the template tag, you first retrieved all the reviews provided by the user:

```
reviews = Review.objects.filter
    (creator__username__contains=username)
```

Since the user field is mapped to the `Review` model as a foreign key, you used the `creator` attribute (which maps to the `User` model) from the `Review` object and then filtered based on the username.

Once you had the list of reviews, you then created a list of books read by the user by iterating upon the objects returned from the database:

```
book_list = [review.book.title for review in reviews]
```

Once this list was generated, you wrote the following line to return a template context object. This is nothing but a dictionary of the `book_list` variable you can now render inside the template:

```
return {'books_read': book_list}
```

In this case, you will be able to access the list of books read by referencing the `books_read` variable inside the template.

3. With the template tag created, you can now move on to create the `book_list` template (if not already created in the previous exercises). If you have created it already, you can skip to *step 4*. For this, create a new file named `book_list.html` under the `templates` directory of the reviews app. Once this file is created, add the following code inside it:

```
<p>Books read</p>
<ul>
  {% for book in books_read %}
  <li>{{ book }}</li>
  {% endfor %}
</ul>
```

This is a plain HTML template, where you render the list of books provided by the custom inclusion tag that we built in *step 2*.

As you can see, you have created a `for` loop that iterates over the `books_read` variable provided by the inclusion tag, and then, for every element inside the `books_read` variable, you create a new list item.

4. With the `book_list` template completed, it's time to integrate the template tag into the user profile page. To do this, open the `profile.html` (responsible for rendering the user profile page) file located under the `templates` directory, present in the root directory of the project, and make the following changes:

Find the following command:

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

After the preceding command, you need to load the template tag by adding the following statement:

```
{% load profile_tags %}
```

This will load the tags from the `profile_tags` file. The order of these statements is important because Django requires that the `extends` tag comes first before any other tag in the template file. Failing to do so will result in a template rendering failure while trying to render the template.

5. Now, with the tag loaded, add the `book_list` tag. For this, replace the code under the `profile.html` file inside the `templates` directory with the following code:

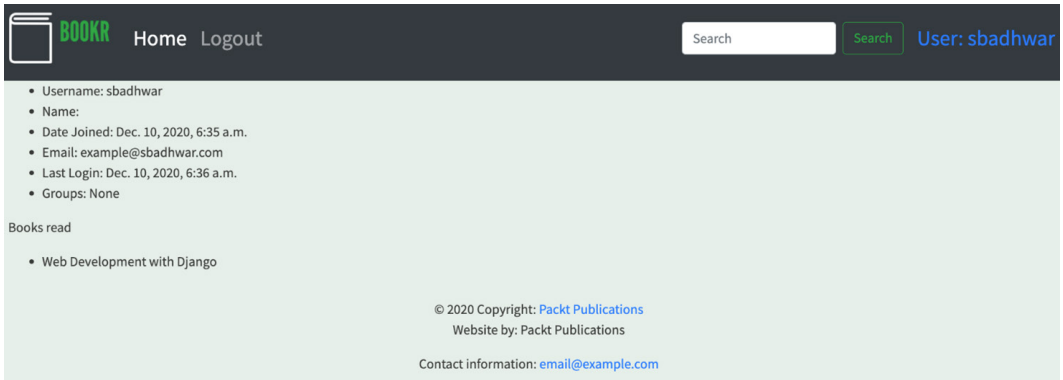
```
{% extends "base.html" %}

{% load profile_tags %}

{% block title %}Bookr{% endblock %}

{% block heading %}Profile{% endblock %}

{% block content %}
  <ul>
    <li>Username: {{ user.username }} </li>
    <li>Name: {{ user.first_name }} {{ user.last_name
      }}</li>
    <li>Date Joined: {{ user.date_joined }} </li>
    <li>Email: {{ user.email }}</li>
    <li>Last Login: {{ user.last_login }}</li>
    <li>Groups: {{ groups }}{% if not groups %}None
      {% endif %} </li>
  </ul>
  {% book_list user.username %}


{% endblock %}
```

The code segment marked in bold instructs Django to use the `book_list` custom tag to render the list of books read by the user.

6. With this, you have successfully built the custom inclusion tag and integrated it with the user profile page. To see how your work renders in the browser, run the following command:

```
python manage.py runserver localhost:8080
```

Then, navigate to your user profile by visiting `http://localhost:8080/`. Once you are on your user profile, if you have added any book to the list of books reviewed, you will see a page that resembles the one shown in the following screenshot:



Activity Figure 11.1: The Bookr user profile page

If you see a page that resembles the one shown in *Activity Figure 11.1*, you have completed the activity successfully.

Chapter 12, Building a REST API

Activity 12.01 – creating an API endpoint for a top contributors page

The following steps will help you complete this activity:

1. Open `bookr/reviews/models.py` and add the following method to the `Contributor` class to count the number of contributions:

```
    def number_contributions(self):
        return self.bookcontributor_set.count()
```

2. Open `bookr/reviews/serializers.py` and add `ContributionSerializer`, which represents a single contribution made by a contributor:

```
from .models import BookContributor

class
ContributionSerializer(serializers.ModelSerializer):
    book = BookSerializer()
    class Meta:
```

```
        model = BookContributor
        fields = ['book', 'role']
```

Here, we use `BookSerializer` from the previous exercise to provide details regarding the specific books. In addition to the `book` field, we add the `role` field, as requested by the frontend developer.

3. Add another serializer to `bookr/reviews/serializers.py` to serialize `Contributor` objects in the database, as follows:

```
from .models import Contributor

class
ContributorSerializer(serializers.ModelSerializer):
    bookcontributor_set = ContributionSerializer
        (read_only=True, many=True)
    number_contributions = serializers.ReadOnlyField()
    class Meta:
        model = Contributor
        fields = ['first_names', 'last_names',
                  'email', 'bookcontributor_set',
                  'number_contributions']
```

There are two things to note regarding this new serializer. The first is that the `bookcontributor_set` field gives us a list of contributions made by a specific contributor. Another point to note is that `number_contributions` uses the method we defined in the `Contributor` class. For this to work, we need to set this as a read-only field. This is because it does not make sense to directly update the number of contributions; instead, you add books to the contributor.

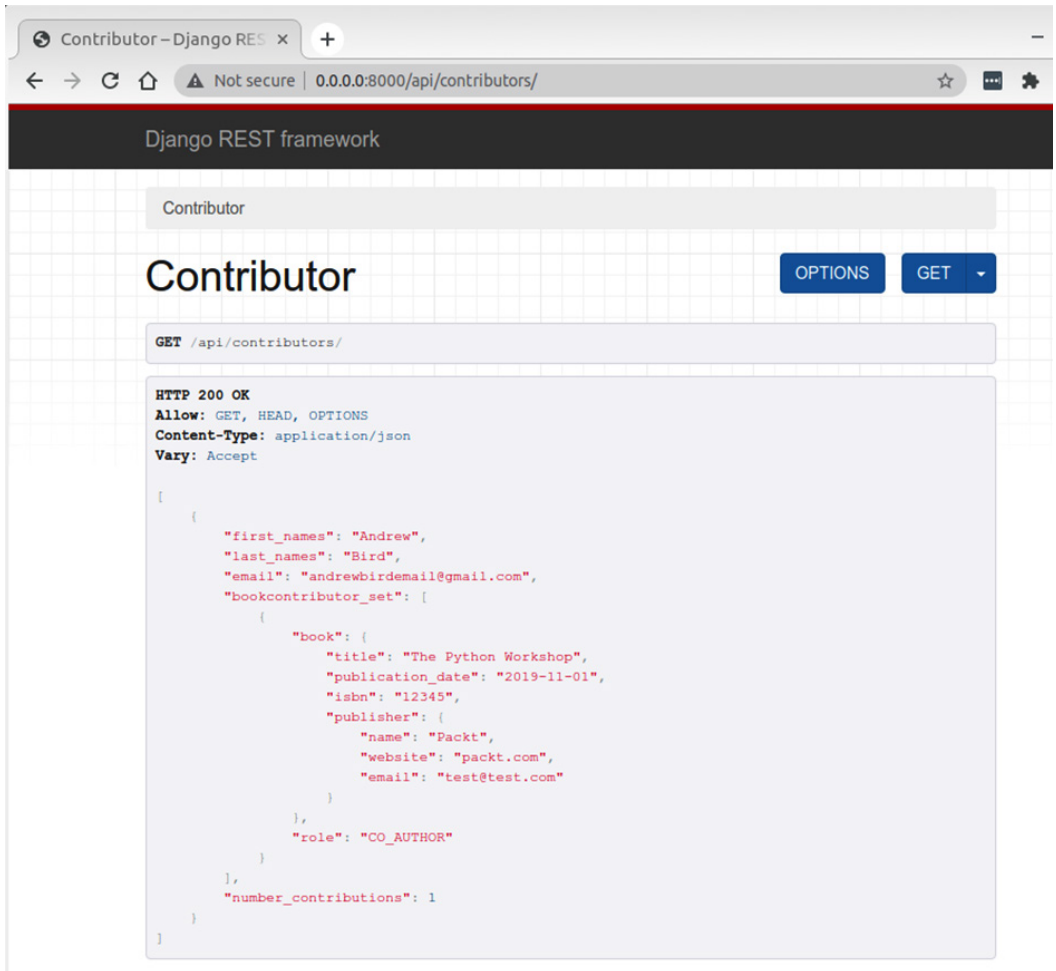
4. Add `ListAPIView` like our existing `AllBooks` view in `api_views.py`:

```
from .models import Contributor
from .serializers import ContributorSerializer

class ContributorView(generics.ListAPIView):
    queryset = Contributor.objects.all()
    serializer_class = ContributorSerializer
```

5. Finally, add a URL pattern in `bookr/reviews/urls.py`:

```
path('api/contributors/'),
path(api_views.ContributorView.as_view()),
path(name='contributors')
```

You should now be able to run a server and access your API view at `http://0.0.0.0:8000/api/contributors/`:



Activity Figure 12.1: Top contributors endpoint

In this activity, you created an API endpoint to provide data to a frontend application. You used class-based views and model serializers to write clean code.

Chapter 13, Generating CSV, PDF, and Other Binary Files

Activity 13.01 – exporting the books read by a user as an XLSLX file

1. Before you can start with the activity, you will require the `XlsxWriter` library to be installed. For this, run the following command:

```
pip install XlsxWriter
```

2. To export the reading history of the user, the first step will be to fetch the reading history for the currently logged-in user. To do this, create a new method named `get_books_read()` inside the `utils.py` file under the `bookr` directory you created as a part of *Exercise 13.06 – visualizing a user's reading history on the user profile page*, as shown in the following code snippet:

```
def get_books_read(username):
    """Get the list of books read by a user.

    :param: str username for whom the
        book records should be returned

    :return: list of dict of books read
        and date of posting the review
    """
    books =
        Review.objects.filter
        (creator__username__contains=username).all()
    return [{'title': book_read.book.title,
             'completed_on': book_read.date_created}
             for book_read in books]
```

In the preceding method, you take in the username of the currently logged-in user for whom the book reading history needs to be retrieved. Based on the username, you filter on the `Review` object, which is used to store the records of the books reviewed by the user.

Once the objects are filtered, the method creates a list of dictionaries, mapping the name of the book and the date on which the user finished reading it, and returns this list.

3. With the helper method created, open the `views.py` file under the `bookr` directory and import the utility method you created in *step 2*, as well as the `BytesIO` library and the `XlsxWriter` package, as shown in the following code snippet:

```
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse
from django.shortcuts import render

from plotly.offline import plot
```

```
import plotly.graph_objects as graphs

from io import BytesIO
import xlsxwriter

from .utils import get_books_read,
get_books_read_by_month
```

4. Once you have set up the required imports, create a new view function named `reading_history()`, as shown in the following code snippet:

```
@login_required
def reading_history(request):
    user = request.user.username
    books_read = get_books_read(user)
```

In the preceding code snippet, you first created a view function named `reading_history`, which will be used to serve the requests to export the reading history of the currently logged-in user as an XLSX file. This view function is decorated with the `@login_required` decorator, which enforces that only logged-in users can access this view inside the Django application.

Once the username is obtained, you then pass the username to the `get_books_read()` method created in *step 2* to obtain the reading history of the user.

5. With the reading history obtained, you now need to build an XLSX file with this reading history. For this use case, there is no need to create a physical XLSX file on disk; you can use an in-memory file. To create this in-memory XLSX file, add the following code snippet to the `reading_history` function:

```
temp_file = BytesIO()

workbook = xlsxwriter.Workbook(temp_file)
worksheet = workbook.add_worksheet()
```

In the preceding code snippet, you first created an in-memory binary file using the `BytesIO()` class. The object returned by the `BytesIO()` class represents an in-memory file and supports all the operations that you would do on a regular binary file in Python.

With the file object in place, you passed this file object to the `Workbook` class of the `xlsxwriter` package to create a new workbook that will store the reading history of the user.

Once the workbook was created, you added a new worksheet to the `Workbook` object, which helped organize your data into a row-and-column format.

6. With the worksheet created, you can now parse the data you obtained as a result of the `get_books_read()` method call from *step 4*. To do this, add the following code snippet to your `reading_history()` function:

```
data = []
for book_read in books_read:
    data.append([book_read['title'],
                 str(book_read['completed_on'])])

for row in range(len(data)):
    for col in range(len(data[row])):
        worksheet.write(row, col, data[row][col])

workbook.close()
```

In the preceding code snippet, you first created an empty list object to store the data to be written to the XLSX file. The object is populated by iterating over the `books_read` list of dictionaries and formatting it such that it represents a list of lists, where every element in the sub-list is a value for a specific column.

Once this list of lists was created, you then iterated over it to write the values for the individual columns present in a specific row.

Once all the values were written, you then went ahead and closed the workbook to make sure all the data was written correctly, and no data corruption occurred.

7. With the data now present in the in-memory binary file, you need to read that data and send it as an HTTP response to the user. To do this, add the following code snippet to the `reading_history()` view function:

```
data_to_download = temp_file.getvalue()

response =
HttpResponse(content_type='application/vnd.ms-excel')
response['Content-Disposition'] = 'attachment;
filename=reading_history.xlsx'
response.write(data_to_download)

return response
```

In the preceding code snippet, you first retrieved the data stored inside the in-memory binary file by using the `getvalue()` method of the in-memory file object.

You then prepared an HTTP response with the `ms-excel` content type and indicated that this response should be treated as a downloadable file, by setting the `Content-Disposition` header.

Once the header was set up, you then wrote the content of the in-memory file to the response object and then returned it from the view function, essentially starting a file download for the user.

8. With the view function created, the last step is to map this view function to a URL. To do this, open the `urls.py` file under the `bookr` application directory and add the bold code in the following code snippet to the file:

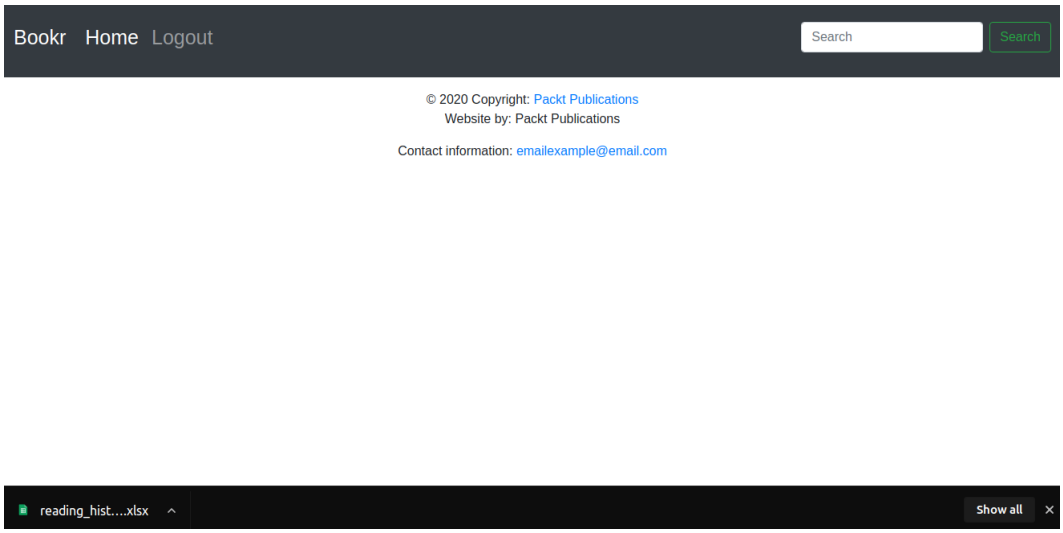
```
import books.views

urlpatterns = [...,
    path('accounts/profile/reading_history'),
        (bookr.views.reading_history),
        (name='reading_history'),
...]
```

9. The URL is mapped. Now, start the application by running the following command:

```
python manage.py runserver localhost:8080
```

After running the command, visit `http://localhost:8080/accounts/profile/reading_history`.

If you are not already logged in, you will be redirected to the login page, and after a successful login, you will be prompted to accept a file download with the name `reading_history.xlsx`:



Activity Figure 13.1: The reading_history file being downloaded

If you see a file downloading, similar to what you can see in *Activity Figure 13.1*, you have successfully completed the activity.

> **Note**
> If you don't have any reading history associated with your user account, the downloaded file will be a blank Excel file.

Chapter 14, Testing

Activity 14.01 – testing models and views in bookr

1. To start with, you will write test cases for the components of the `review` module that you have built so far. First, create a directory named `tests/` inside the `reviews` application directory, which you will use to store the test cases for different components of the `reviews` application. This can be done by running the following command inside the `reviews` application directory:

   ```
   mkdir tests
   ```

   Once this is done, you need to make sure that Django recognizes this directory as a module and not as a regular directory. To make this happen, create an empty file with the name `__init__.py` under the `tests` directory you created just now by running the following command:

   ```
   touch __init__.py
   ```

   Once this is done, you can remove the `tests.py` file from the `reviews` directory, since we are now moving toward a pattern of using modularized test cases.

2. Once the `tests` directory is created, it is time to write test cases for the components, starting with the writing of test cases for the models inside the `reviews` application. To do this, create a new file named `test_models.py` under the `tests` directory you created in *step 1* by running the following command:

   ```
   touch test_models.py
   ```

   Once the file is created, add the following code to it:

test_models.py

```
from django.test import TestCase

from reviews.models import Book, Publisher,
Contributor


class TestPublisherModel(TestCase):
    def test_create_publisher(self):
```

```
            publisher = Publisher.objects.create
                (name='Packt', website='www.packt.com',
                email='contact@packt.com')
            self.assertIsInstance(publisher, Publisher)
```

You can find the complete code for this file at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter14/Activity14.01/reviews/tests/test_models.py`.

In the preceding code snippet, you have written a couple of test cases for the models that you created for the `reviews` application.

You started by importing Django's `TestCase` class and the models you are going to test in this exercise:

```
from django.test import TestCase

from reviews.models import Book, Publisher,
Contributor
```

With the required classes imported, you defined the test cases. To test the `Publisher` model, you first created a new class, `TestPublisherModel`, which inherits from Django's `TestCase` class. Inside this class, you added the following test, which checks whether the `Publisher` model objects are being created successfully or not:

```
def test_create_publisher(self):
    publisher = Publisher.objects.create
        (name='Packt',website='www.packt.com',
        email='contact@packt.com')
    self.assertIsInstance(publisher, Publisher)
```

This validation is performed by calling the `assertIsInstance()` method on the `publisher` object to validate its type. Following a similar pattern, you also created test cases for the `Contributor` and `Book` models. Now, you can go ahead and create test cases for the remaining models of the `reviews` application.

3. With the test cases for models now written, the next step is to cover test cases for our views. To do this, create a new file named `test_views.py` under the `tests` directory of the `reviews` application directory by running the following command from inside the `tests` directory:

```
touch test_views.py
```

> **Note**
>
> On Windows, you can create this file using Windows Explorer.

Once the file is created, write the test cases inside it. For testing, use Django's `RequestFactory` to test the views. Add the following code to your `test_views.py` file:

```
from django.test import TestCase, RequestFactory

from reviews.views import index


class TestIndexView(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    def test_index_view(self):
        request = self.factory.get('/index')
        request.session = {}
        response = index(request)
        self.assertEquals(response.status_code, 200)
```

Let us examine the code in detail. In the first two lines, you imported the required classes to write our test cases, including `TestCase` and `RequestFactory`. Once the base classes were imported, you then imported the `index` method from the `reviews.views` module to be tested. Next up, you created `TestCase` by creating a new class named `TestIndexView`, which will encapsulate the test cases for the view functions. Inside this `TestIndexView`, you added the `setUp()` method, which will help you create a request factory instance for use in every test case:

```
def setUp(self):
    self.factory = RequestFactory()
```

With the `setUp()` method defined, you wrote a test case for the `index` view. Inside this test case, you first created a `request` object as if you were trying to make an `HTTP GET` call to the `'/index'` endpoint:

```
request = self.factory.get('/index')
```

Once the `request` object is available, you set the `session` object to point to an empty dictionary, since you currently do not have a session available:

```
request.session = {}
```

With the `session` object now pointing to an empty dictionary, you can now test the `index` view function by passing the `request` object to it as follows:

```
response = index(request)
```

Once the response is generated, you validate the response by checking the status code of the response using the `assertEquals` method:

```
self.assertEquals(response.status_code, 200)
```

4. With the test cases now in place, run and check whether they pass successfully. To do this, run the following command:

```
python manage.py test
```

Once the command finishes, you should expect to see the following output generated:

```
% python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.........
----------------------------------------------------------------------
-------
Ran 9 tests in 0.290s
```

In the preceding output, you can see that all the test cases that you implemented have passed successfully, validating that the components work in the way they are expected to.

Chapter 15, Django Third-Party Libraries

Activity 15.01 – using FormHelper to update forms

The following steps will help you complete this activity:

1. Open the `reviews` app's `forms.py`. Create a class called `InstanceForm`. It should inherit from `forms.ModelForm`:

```
class InstanceForm(forms.ModelForm):
```

It should be the first class defined in the file, as other classes will inherit from it.

2. At the start of the file, you should already be importing the `FormHelper` class from `crispy_forms.helper`:

```
from crispy_forms.helper import FormHelper
```

Add the `__init__` method to `InstanceForm`. It should accept `*args` and `**kwargs` as arguments and pass them through to `super().__init__()`:

```
class InstanceForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Then, instantiate a `FormHelper` instance and set it to `self.helper`:

```
        self.helper = FormHelper()
```

This is the helper object that `django-crispy-forms` will query to find the form attributes. Since this form is to be submitted with `POST`, which is the default method of `FormHelper`, we don't need to set any attributes.

3. The `Submit` class must first be imported from `crispy_forms.layout` (you should already have this from *Exercise 15.04 – Using Django crispy forms with SearchForm*):

```
from crispy_forms.layout import Submit
```

Next, we need to determine the submit button's title. Inside the `InstanceFrom__init__` method, check whether `kwargs` contains an `instance` item. If it does, then `button_title` should be `Save` (as we are saving the existing instance). Otherwise, `button_title` should be `Create`:

```
        if kwargs.get("instance"):
            button_title = "Save"
        else:
            button_title = "Create"
```

Finally, create a `Submit` button, using `button_title` as its second argument, and then pass this to the `add_input` method of `FormHelper`:

```
        self.helper.add_input(Submit("",
        button_title))
```

This will create the submit button for `django-crispy-forms` to add to the form.

4. Change `PublisherForm`, `ReviewForm`, and `BookMediaForm` in this file so that instead of inheriting from `models.ModelForm`, they inherit from `InstanceForm`.

Go to the following line:

```
class PublisherForm(models.ModelForm):
```

Change it to this:

```
class PublisherForm(InstanceForm):
```

Next, go to the following line:

```
class ReviewForm(models.ModelForm):
```

Change it to this:

```
class ReviewForm(InstanceForm):
```

Finally, for `BookMediaForm`, go to the following line:

```
class BookMediaForm(models.ModelForm):
```

Change it to this:

```
class BookMediaForm(InstanceForm):
```

No other lines need to be changed. You can then save and close `forms.py`.

5. Open `instance-form.html` in the `reviews` app's `templates` directory. First, you must use the `load` template tag to load `crispy_forms_tags`. Add this on the second line of the file:

```
{% load crispy_forms_tags %}
```

Then, scroll to the bottom of the file where the `<form>` tag is located. You can delete the entire form code, including the opening `<form>` and closing `</form>` tags. Then, replace it with the following:

```
{% crispy form %}
```

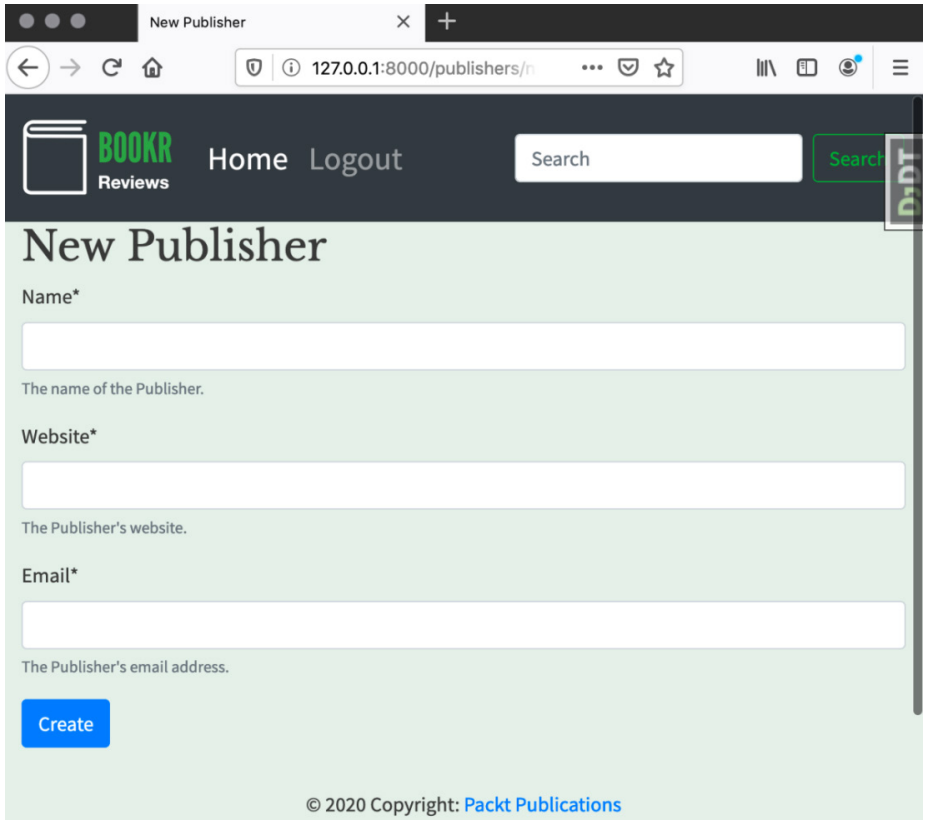
This will render `form`, including the `<form>` tags, CSRF token, and submit button. You can save and close `instance-form.html`.

6. Open the `reviews` app's `views.py`. In the `book_media` view function, locate the call to the `render` function. In the context dictionary, remove the `is_file_upload` item. The render call line code should then be like this:

```
return render(request,
    "reviews/instance-form.html",
    {"instance": book,
    "form": form, "model_type": "Book"})
```

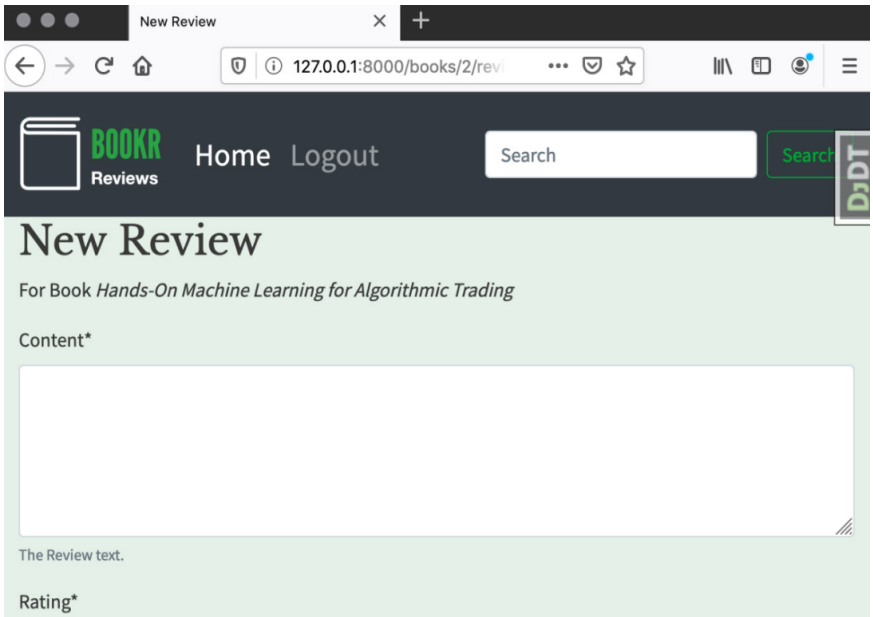
We no longer need `is_file_upload`, as the `crispy` template tag will automatically render the form with the correct `enctype` attribute if the form contains the `FileField` or `FormField` fields. Save `views.py`.

You can now start the Django development server and try using the **Publisher**, **Review**, and **Book Media** forms. They should look nice, since they use the proper Bootstrap classes. The submit buttons should automatically update, based on the create or edit mode you are in for that form. Verify that you can update book media, which will indicate that `enctype` is being set properly on the form:

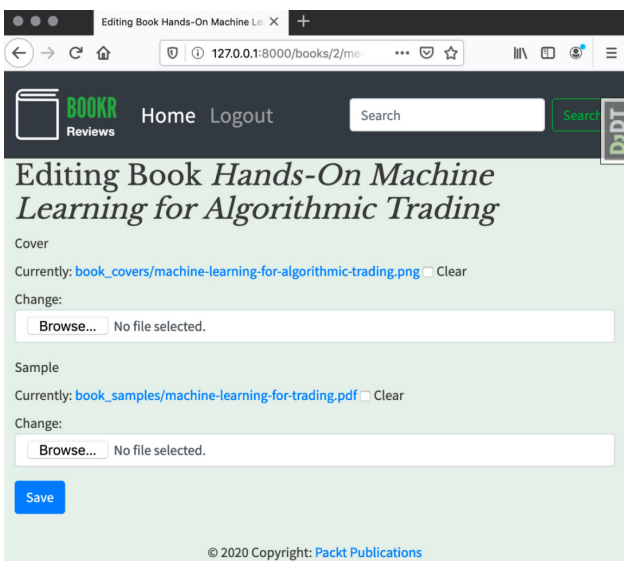


Activity Figure 15.1: The New Publisher page

The **New Review** form and the **Book Media** page should appear as follows:



Activity Figure 15.2: The New Review form



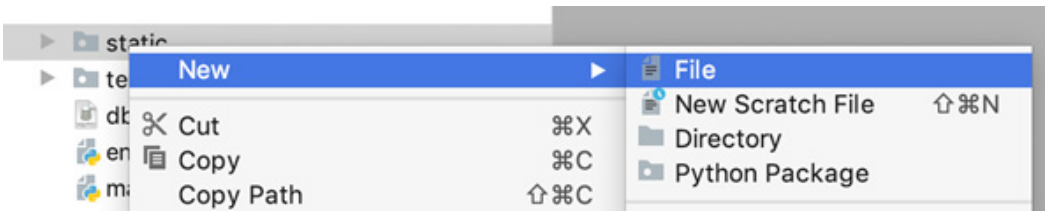
Activity Figure 15.3: The Book Media page

Chapter 16, Using a Frontend JavaScript Library with Django

Activity 16.01 – reviews preview

The following steps will help you complete this activity:

1. Delete the `react_example` view from `views.py`. Remove the `react-example` URL map from `urls.py`, and then delete the `react-example.html` template file and `react-example.js` JavaScript file.
2. In the project's `static` directory, create a new file named `recent-reviews.js`:



Activity Figure 16.1: Create a JavaScript file in the main static directory

3. Define the `ReviewDisplay` class like this:

```
class ReviewDisplay extends React.Component {
  constructor(props) {
    super(props);
    this.state = { review: props.review };
  }
}
```

Here, the `state` attribute is created with `review` that will be passed in using the `review` attribute.

4. Implement the `render` method like this:

```
    render () {
      const review = this.state.review;

      return <div className="col mb-4">
      <div className="card">
      <div className="card-body">
      <h5 className="card-title">{ review.book }
      <strong>({ review.rating })</strong>
        </h5>
      <h6 className="card-subtitle mb-2 text-muted">
```

```
          { review.creator.email }</h6>
        <p className="card-text">{ review.content }</p>
        </div>
        <div className="card-footer">
        <a href={'/books/' + review.book_id + '/' }
          className="card-link">View Book</a>
        </div>
        </div>
        </div>;
      }
```

Note that it is not necessary to create a `review` alias variable; the data can be accessed through `state` throughout the HTML. For example, `{ review.book }` can be replaced with `{ this.state.review.book }`. The completed class should now look like this: `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter16/Activity16.01/bookr/static/recent-reviews.js`.

5. Create a `RecentReviews` class, with a `constructor` method like this:

```
class RecentReviews extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            reviews: [],
            currentUrl: props.url,
            nextUrl: null,
            previousUrl: null,
            loading: false
        };
    }
}
```

This reads `currentUrl` from the one provided in `props`, and other values are set as defaults.

6. Create a `fetchReviews` method that returns immediately if a load is in process:

```
fetchReviews() {
  if (this.state.loading)
      return;
this.setState( {loading: true} );
}
```

7. After setting the state in the previous step, add the call to the `fetch` function:

```
fetch(this.state.currentUrl, {
  method: 'GET',
  headers: {
      Accept: 'application/json'
  }
}).then((response) => {
  return response.json()
}).then((data) => {
this.setState({
    loading: false,
    reviews: data.results,
    nextUrl: data.next,
    previousUrl: data.previous
  })
})
```

This will fetch `currentUrl`, and then update `nextUrl` and `previousUrl` to the values of `next` and `previous` that the Django server generates. We also set our `reviews` to the `data.results` we retrieved and set `loading` back to `false`, since the load is complete.

8. The `componentDidMount` method simply calls the `fetchReviews` method:

```
componentDidMount() {
  this.fetchReviews()
}
```

This method is called by React when the component first renders.

9. The `loadNext` method should return if there is no `nextUrl`. Otherwise, set `currentUrl` as `nextUrl`. Then, call `fetchReviews`:

```
loadNext() {
  if (this.state.nextUrl == null)
      return;

  this.state.currentUrl = this.state.nextUrl;
  this.fetchReviews();
}
```

10. `loadPrevious` is like `loadNext`, with `previousUrl` used in place of `nextUrl`:

```
loadPrevious() {
  if (this.state.previousUrl == null)
      return;

  this.state.currentUrl = this.state.previousUrl;
```

```
    this.fetchReviews();
  }
```

11. The `render` method is quite long. We will look at it in chunks. First, it should return some loading text if `state.loading` is true:

```
  if (this.state.loading) {
    return <h5>Loading...</h5>;
  }
```

12. Still inside the `render` method, we'll now create the `previous` and `next` buttons and assign them to the `previousButton` and `nextButton` variables respectively. Their `onClick` action is set to trigger the `loadPrevious` or `loadNext` method. We can disable them by checking whether `previousUrl` or `nextUrl` is `null`. Here is the code:

```
          const previousButton = <button
              className="btn btn-secondary"
              onClick={ () => { this.loadPrevious() } }
              disabled={ this.state.previousUrl == null
              }>Previous</button>;

          const nextButton = <button
              className="btn btn-secondary float-right"
              onClick={ () => { this.loadNext() } }
              disabled={ this.state.nextUrl == null }>
              Next</button>;
```

Note that `nextButton` has the additional `float-right` class, which displays it on the right of the page.

13. Next, we define the code that displays a list of `ReviewDisplay` elements:

```
  if (this.state.reviews.length === 0) {
  reviewItems = <h5>No reviews to display.</h5>
  } else {
  reviewItems = this.state.reviews.map((review) => {
      return <ReviewDisplay key={review.pk}
          review={review}/>
      })
  }
```

If the length of the `reviews` array is `0`, then the content to display is set to `<h5>No reviews to display.</h5>`. Otherwise, we iterate over the `reviews` array using its `map` method. This will build an array of `ReviewDisplay` elements. We give each of these a unique `key` of `review.pk`, and also pass in `review` itself as a property.

14. Finally, all the content we built is bundled together inside some `<div>` instances and returned, as per the example given:

```
return <div>
<div className="row row-cols-1 row-cols-sm-2 row-cols-md-3">
  { reviewItems }
</div>
<div>
  {previousButton}
  {nextButton}
</div>
</div>;
```

The finished `RecentReviews` class is at `https://github.com/PacktPublishing/Web-Development-with-Django-Second-Edition/blob/main/Chapter16/Activity16.01/bookr/static/recent-reviews.js`.

You can save and close `recent-reviews.js`.

15. Open the project's `base.html` (not the review-specific one). Between the `{% block content %}` and `{% endblock %}` template tags, after the closing `{% endif %}` containing the previously viewed book history, add the `Recent Reviews` heading:

```
<h4>Recent Reviews</h4>
```

16. Underneath `<h4>` added in the previous step, create a `<div>` element, with a unique `id` attribute:

```
<div id="recent_reviews"></div>
```

In this example, we are using `id` of `recent_reviews`, but yours could be different. Just make sure you use the same ID when referring to the `<div>` element in *step 18*.

17. Under the `<div>` element that was just added, but still before `{% endblock %}`, add your `<script>` elements:

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.
development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
<script crossorigin src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
<script src="{% static 'recent-reviews.js' %}" type="text/
babel"></script>
```

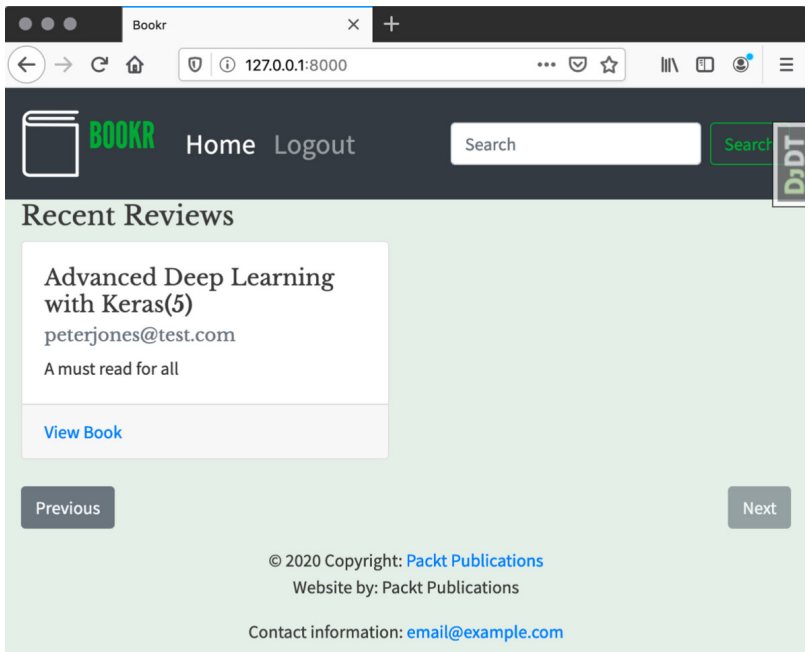
18. Finally, add another `<script>` element to render the React component:

```
<script type="text/babel">
  ReactDOM.render(<RecentReviews url="{% url
  'api:review-list' %}?limit=6" />,
  document.getElementById('recent_reviews')
```

```
    );
</script>
```

The `url` that is passed into the component's `prop` is generated by Django's `url` template tag. We manually append the `?limit=6` argument to limit the number of reviews that are returned. In the `document.getElementById` call, make sure you use the same ID string that you gave to your `<div>` in *step 16*.

Start the Django development server if it is not already running, and then go to `http://127.0.0.1:8000/` (the Bookr main page). You should see the first six reviews and be able to page through them by clicking the **Previous** and **Next** buttons:



Activity Figure 16.2: The last page of reviews