Django In Action

Chapter 5. Sample exercise solutions

Throughout the book there have been exercises to help you practice what you've learned. This appendix contains solutions to the exercises in case you get stuck, or if you want to compare your solution to my own. Coding is like writing; everyone has their own style, so if your code worked, it's right, but it's always useful to see how others approach a design in order to learn.

5.1. Chapter 2

Chapter 2 is all about getting started, building your first Django project and an app with a view to go inside it.

Exercise 1

Add an About page to the site using the same techniques as you did for the credits page. Instead of using plain text, have the content block contain valid HTML content.

To build the About page, you need to create a new view function, which can be seen in listing E.1.

Listing E. 1. The view function for the About page

```
# RiffMates/home/views.py
from django.http import HttpResponse
...

def about(request):
    content = [ ①
        "<!doctype html>",
        '<html lang="en">',
        "<head>",
        "  <title>RiffMates About</title>",
        "</head>",
        "<body>",
        "  <h1>RiffMates About</h1>",
        "  <p>",
        "   RiffMates is a for musicians seeking musicians. Find your next ",
        "   band or band-mate. Find your next gig.",
        "  </p>",
        "</body>",
        "</html>",
    ]

    content = "\n".join(content) ②
    return HttpResponse(content) ③
```

① The HTML content to be displayed in the page, stored in a list

② Use `"\n".join()` to merge the list into a single text value

③ Django expects a view to return an `HttpResponse` object, which takes the contents of the page to be displayed

You associate the view with a URL by registering it with Django as in listing E.2.

Listing E. 2. Register the About view as a route

```
# RiffMates/RiffMates/urls.py
...
from home import views as home_views

urlpatterns = [
    # ...
    path("about/", home_views.about),
    # ...
]
```

Exercise 2

Add a Version Info page that contains the version number of your site. Instead of using plain text, return valid JSON. You can do this by adapting the MIME type or by using a `django.http.JsonResponse` object instead. Details on the `JsonResponse` object can be found in the documentation: https://docs.djangoproject.com/en/dev/ref/request-response/#jsonresponse-objects

Django provides the `JsonReponse` class, which inherits from `HttpResponse` so that you can return JSON data from a view. `JsonReponse` expects a value that can be serialized to JSON and it sets the appropriate MIME type for you. A view with the version info of `"0.0.1"` is shown in listing E.3.

Listing E. 3. The Version Info view which returns JSON

```
# RiffMates/home/views.py
from django.http import HttpResponse, JsonResponse
...
def version(request):
    data = { ①
        "version": "0.0.1",
    }

    return JsonResponse(data) ②
```

① Version data to be serialized to JSON

② The `JsonResponse` object takes data to be serialized and sets the correct MIME type for JSON content

As with all views, Version Info must be registered against a URL. This is shown in E.4.

Listing E. 4. Register the Version Info view as a route

```
# RiffMates/RiffMates/urls.py
...
from home import views as home_views

urlpatterns = [
    # ...
    path("version/", home_views.version),
    # ...
]
```

5.2. Chapter 3

In chapter 3 you learned about Django's template engine, including how to use tags and filters, as well as how Django escapes content to make it safe to render in HTML.

Exercise 1

Create a new base HTML file using your favorite web-style framework such as Bootstrap or Tailwind. Once you have something pretty, test it out by changing 'news2.html' to extend from it.

Django's template engine allows you to compose templates together to promote re-use of HTML or any other text content. One of these re-use mechanisms is inheritance. As HTML is quite verbose, writing a good base file to inherit from once means your individual pages can be quite small.

Listing E.5 contains the top part of the content from a base HTML page that uses the Bootstrap 5 framework.

Listing E. 5. Declaration and head part of a base HTML file

```
<!-- RiffMates/templates/base_bootstrap.html -->
<!doctype html>
<html lang="en" class="h-100">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <meta name="description" content="">
  <title>
    {% block title %} ①
      RiffMates {% if title %}&mdash; {{ title }} {% endif %}
    {% endblock %}
  </title>

  <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/
[CA]css/bootstrap.min.css"
```

```
    rel="stylesheet"
    integrity="sha384-GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/
[CA]Jr59b6EGGoI1aFkw7cmDA6j6gD"
    crossorigin="anonymous"> ②

<style> ③
  main > .container {
    padding: 60px 15px 0;
  }
</style>

</head>
<body class="d-flex flex-column h-100"> ④
```

① Putting the page's title in a block allows it to be overridden by child pages

② Link to the Bootstrap 5 stylesheet

③ A little extra style definition to leave space at the top of the page for the navbar

④ These class declarations change how the layout of the page is handled

RiffMates has a navigation bar, so the next piece of content in the base file (listing E.6) uses Bootstrap's `<nav>` tag.

Listing E. 6. Bootstrap navbar for the base file

```
<header>
  {% block navbar %} ①
    <!-- Fixed navbar -->
    <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">RiffMates</a>
        <button class="navbar-toggler" type="button" data-bs-toggle=
[CA]"collapse" data-bs-target="#navbarCollapse" aria-controls=
[CA]"navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarCollapse">
          <ul class="navbar-nav me-auto mb-2 mb-md-0">
            <li class="nav-item">
              <a class="nav-link active" aria-current="page"
                href="{% url 'news' %}">News</a>
            </li>
            <li class="nav-item">
              <a class="nav-link active" aria-current="page"
                href="{% url 'credits' %}">Credits</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  {% endblock navbar %}
```

```
</header>
```

① Putting the content in a block allows it to be overridden or added to by inheriting pages

The last portion of the new base file contains a block for page content, a footer, and the JavaScript expected by the Bootstrap framework. Combine listing E.7 with the other three parts to create your new base file.

Listing E. 7. Section where main content gets rendered

```
<!-- Begin page content -->
<main class="flex-shrink-0">
  <div class="container">
    {% block content %} ①
    {% endblock content %}
  </div>
</main>

<footer class="footer mt-auto py-3 bg-light">
  <div class="container">
    {% block footer %} ②
      <span class="text-muted">RiffMates: by musicians for musicians</span>
    {% endblock footer %}
  </div>
</footer>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/
[CA]bootstrap.bundle.min.js" integrity="sha384-w76AqPfDkMBDXo30jS1Sgez6pr
[CA]3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBTkF7CXvN" crossorigin="anonymous">
[CA]</script>

</body>
</html>
```

① This empty block is where page content goes in inheriting files

② Putting the footer in its own block allows the content to be overridden or added to in child pages

With the new base_boostrap.html file in place, you can take advantage of the sexier look by inheriting from it. To try it out, modify the news2.html file, changing the `extends` statement to use your new base file. The result is shown in listing E.8.

Listing E. 8. Modified news2.html file using the Bootstrap base template

```
<!-- RiffMates/templates/news2.html -->
{% extends "base_bootstrap.html" %} ①

{% block title %}
  {{block.super}}: News
{% endblock %}
```

```
{% block content %}
  <h1>RiffMates News (2)</h1>

  <ul>
    {% for item in news %}
      <li>{{ item }}</li>
    {% endfor %}
  </ul>

{% endblock content %}
```

① Changing the `extends` tag is all that is required to change where this template is inherited from

Exercise 2

Write a more advanced version of the News page by adding a `news_advanced()` view to `home/views.py`. Each data item in this view is a tuple containing the date and subject of the news. Create a news_adv.html template that shows the data in a table. Look-up the `{% cycle %}` tag and the `|date` filter in Appendix C and apply them to your table, using `{% cycle %}` to apply striping to the table and `|date` to format the news item's date.

This exercise is mostly about getting more practice with some tags and filters, but to do that you need a view to render your new template. The advanced version of the News page requires a view containing tuples, where each tuple has a date and a news headline. Listing E.9 defines a view that uses today, yesterday, and the day before as dates and three bits of news to go with them.

Listing E. 9. An advanced version of a news view

```
# RiffMates/home/views.py
from datetime import date, timedelta ①

from django.http import HttpResponse, JsonResponse
from django.shortcuts import render

def news_advanced(request):
    today = date.today() ②
    before1 = today - timedelta(days=1) ③
    before2 = today - timedelta(days=2)

    data = { ④
        "news": [
            (today, "Advanced news added! Even more exclamation points!!!"),
            (before1, "RiffMates now has a news page!"),
            (before2, "RiffMates has its first web page"),
        ],
    }

    return render(request, "adv_news.html", data) ⑤
```

① Python's `date` has contains the `today()` method that returns today's date, while `timedelta` allows you to do math on `date` objects

② Get today's date

③ Calculate the day before today

④ Context data for the template engine containing news items for today and the two prior days

⑤ You learned about the `render()` shortcut in chapter 3; it loads and renders a template, returning an `HttpResponse` object with the resulting content

Listing E.10 shows the new view being registered as a route.

Listing E. 10. Register the advanced news view

```
# RiffMates/RiffMates/urls.py
...
from django.urls import path
from home import views as home_views

urlpatterns = [
    # ...
    path("adv_news/", home_views.news_advanced, name="adv_news"),
]
```

The template for the advanced news page iterates over the `"news"` list in the context, rendering a table. Each item of news is a tuple, with the first item in the tuple being the date and the second being the headline. The `{% cycle %}` tag cycles through the values inside of it for each iteration a loop, in this example changing the background color of the table row from a darker to lighter grey. The `| date` filter is similar to the `strftime()` function in Python, formatting a date object for display. The whole template for the advanced news page is in listing E.11.

Listing E. 11. The template for rendering the advanced news page

```
<!-- RiffMates/templates/adv_news.html -->
{% extends "base.html" %}

{% block title %}
  {{block.super}}: News
{% endblock %}

{% block content %}
  <h1>RiffMates Advanced News</h1>

  <table>
    {% for item in news %} ①
      <tr style="background-color: {% cycle '#ddd' '#bbb' %}"> ②
        <td> {{ item.0 | date:"Y-m-d" }} </td> ③
        <td> {{ item.1 }} </td> ④
      </tr>
    {% endfor %}
```

```
  </table>

{% endblock content %}
```

① Iterate over the value of `"news"` in the context dictionary

② The `{% cycle %}` returns the next value in its sequence each time the loop is executed

③ Use the `| date` filter to format the date in year-month-day format

④ Put the news headline in the second column in the row

5.3. Chapter 4

Django's ORM allows you to use a database without writing any SQL. Chapter 4 introduced you to writing `Model` classes, running queries, and accessing parts of a table at a time through pagination.

Exercise 1

Modify the `musicians` index view to take an additional query parameter that specifies the number of objects per page. Make sure to support reasonable default and maximum values.

The `Paginator` class takes the number of items per page in its constructor. Prior to this exercise, this value was hard coded. All three exercises in this chapter use a `Paginator`, so in addition to adding a query parameter specifying the number of items per page, the bounds-checking on this query parameter and the page number query parameter have been moved into utility functions. The two new functions and the new version of `musicians()` is shown in listing E.12.

Listing E. 12. Utility functions for validating the query parameters and an updated `musicians` view

```
# RiffMates/bands/views.py
from bands.models import Band, Musician, Room, Venue
from django.core.paginator import Paginator
from django.shortcuts import get_object_or_404, render

def _get_items_per_page(request): ①
    # Determine how many items to show per page, disallowing <1 or >50
    items_per_page = \
        int(request.GET.get("items_per_page", 10)) ②
    if items_per_page < 1: ③
        items_per_page = 10
    if items_per_page > 50:
        items_per_page = 50

    return items_per_page

def _get_page_num(request, paginator): ④
    # Get current page number for Pagination, using reasonable defaults
    page_num = int(request.GET.get("page", 1)) ⑤
```

```
    if page_num < 1:
        page_num = 1
    elif page_num > paginator.num_pages:
        page_num = paginator.num_pages

    return page_num

def musicians(request): ⑥
    all_musicians = Musician.objects.all().order_by("last_name")
    items_per_page = _get_items_per_page(request) ⑦
    paginator = Paginator(all_musicians, items_per_page)
    page_num = _get_page_num(request, paginator) ⑧
    page = paginator.page(page_num)

    data = {
        "musicians": page.object_list,
        "page": page,
    }

    return render(request, "musicians.html", data)
```

① Shared function to get the query parameter that stores the number of items to display on the page

② Query parameters are strings; convert the value to an integer and set a default value of 10

③ Validate that the number of items per page is an acceptable value

④ Shared function to get the query parameter that stores the page number to be displayed

⑤ Default to page 1, convert the parameter to an integer

⑥ An updated version of `musicians()` that uses the new shared functions

⑦ Call the shared function to get the number of items per page

⑧ Call the shared function to get the page number

Exercise 2

Using the musician index and detail pages as a reference, create similar pages for your `Band` objects. Inside the template, use the `musicians` relationship to show the members of the band.

Like with your musician views, here you need a view to display a single `Band` object and one for listing all `Band` objects. Listing E.13 shows these two views, taking advantage of the shared functions built in exercise 1.

Listing E. 13. Listing and detail views for your bands

```
# RiffMates/bands/views.py
...
def band(request, band_id): ①
    data = {
```

```
        "band": get_object_or_404(Band, id=band_id), ②
    }

    return render(request, "band.html", data)


def bands(request):
    all_bands = Band.objects.all().order_by("name") ③
    items_per_page = _get_items_per_page(request) ④
    paginator = Paginator(all_bands, items_per_page) ⑤

    page_num = _get_page_num(request, paginator) ⑥

    page = paginator.page(page_num) ⑦

    data = {
        "bands": page.object_list,
        "page": page,
    }

    return render(request, "bands.html", data)
```

① The band detail view takes the ID of a `Band` object as an argument

② The `get_object_or_404()` shortcut fetches the `Band` with the given ID or raises an `HTTP404` error

③ The band listing page queries all the `Band` objects

④ Call the shared function to get the number of items per page

⑤ Create a `Paginator` to paginate the list of bands

⑥ Call the shared function to get the page number

⑦ Create a `page` object to be used during rendering that encapsulates the subset of bands to display on this page

Listing E.14 contains the template for the band details page.

Listing E. 14. Template for the band details page

```
<!-- RiffMates/templates/band.html -->
{% extends "base.html" %}

{% block title %}
  {{block.super}}: Band Details
{% endblock %}

{% block content %}
  <h1>{{band.name}}</h1>

  <h2>Band Members</h2>
  <ul>
    {% for musician in band.musicians.all %}
```

```
      <li> <a href="{% url 'musician' musician.id %}">
        {{musician.last_name}}, {{musician.first_name}}
      </a> </li>
    {% empty %}
      <li> <i>No members</i> </li>
    {% endfor %}
  </ul>
{% endblock content %}
```

Listing E.15 contains the template for the bands listing page.

Listing E. 15. Template for the bands listing page

```
<!-- RiffMates/templates/bands.html -->
{% extends "base.html" %}

{% block title %}
  {{block.super}}: Band Listing
{% endblock %}

{% block content %}
  <h1>Bands</h1>

  <ul>
    {% for band in bands %}
      <li> <a href="{% url 'band' band.id %}"> {{band.name}} </a> </li>
    {% empty %}
      <li> <i>No bands in the database</i> </li>
    {% endfor %}
  </ul>

  {% if page.has_other_pages %} ①
    {% if page.has_previous %} ②
      <a href="{% url 'bands' %}?page={{page.previous_page_number}}"
        > Prev</a> ③
         
    {% endif %}
    {% if page.has_next %} ④
      <a href="{% url 'bands' %}?page={{page.next_page_number}}"> Next</a>
    {% endif %}
  {% endif %}
{% endblock content %}
```

① Conditional section if there are more pages

② Conditional section if there is a page previous to this one

③ Link to the previous page using a query parameter

④ Conditional section if there is a page after this one

Exercise 3

Add a page that lists all `Venue` objects along with their corresponding rooms. You will need to iterate over the `room_set` query manager in the template.

A view for listing venues is similar to those for listing bands and musicians, it is shown in listing E.16.

Listing E. 16. Listing and detail views for your bands

```
# RiffMates/bands/views.py
...
def venues(request):
    all_venues = Venue.objects.all().order_by("name")
    items_per_page = _get_items_per_page(request)
    paginator = Paginator(all_venues, items_per_page)

    page_num = _get_page_num(request, paginator)

    page = paginator.page(page_num)

    data = {
        "venues": page.object_list,
        "page": page,
    }

    return render(request, "venues.html", data)
```

The template for this view (listing E.17 iterates over the subset of venues in the page and uses the `.room_set` query manager attached to the `Venue` object to also iterate over the rooms associated with each venue on the page.

Listing E. 17. Template for the venue listing page

```
<!-- RiffMates/templates/venues.html -->
{% extends "base.html" %}

{% block title %}
  {{block.super}}: Venue Listing
{% endblock %}

{% block content %}
  <h1>Venues</h1>

  <ul>
    {% for venue in venues %} ①
      <li> {{venue.name}} </li>
      <ul>
        {% for room in venue.room_set.all %} ②
```

```
          <li> {{room.name}} </li>
        {% empty %}
          <li> <i>No rooms for this venue</i> </li>
        {% endfor %}
      </ul>
    {% empty %}
      <li> <i>No venues in the database</i> </li>
    {% endfor %}
  </ul>

  {% if page.has_other_pages %} ③
    {% if page.has_previous %}
      <a href="{% url 'venues' %}?page={{page.previous_page_number}}">
[CA] Prev</a>
         
    {% endif %}
    {% if page.has_next %}
      <a href="{% url 'venues' %}?page={{page.next_page_number}}"> Next</a>
    {% endif %}
  {% endif %}
{% endblock content %}
```

① Iterate over the venues listed on this page

② The `.room_set` member of a `Venue` object gives access to rooms related to this venue. In the template language the parenthesis are left off the call to the `.all()` method.

③ Conditional block that handles links to previous and next pages if they exist

5.4. Chapter 5

In chapter 5 you were introduced to the Django Admin, a built-in toolkit for managing all of your ORM models. The Django Admin is built using the same techniques you use in your own projects and is quite customizable.

Exercise 1

Add search capabilities to the `BandAdmin` class.

To add search to a Django `ModelAdmin` class you add the `.search_fields` member to it as shown in listing E.18.

Listing E. 18. A search field for the `BandAdmin` class

```
# RiffMates/bands/admin.py
...
@admin.register(Band)
class BandAdmin(admin.ModelAdmin):
    list_display = ("id", "name", "show_members")
```

```
    search_fields = ("name",) ①
```

① The `Band` class can now be searched based on its `.name` field

Exercise 2

Add default sorting by name to the `Band`, `Venue`, and `Room` classes

The `Meta` inner class is used to change the behavior of your model classes. You use the `ordering` member list to determine default sorting behavior. Listing E.19 shows the `Meta` inner class on the `Band`, `Venue`, and `Room` classes.

Listing E. 19. Adding default ordering to a model

```
# RiffMates/bands/models.py
...
class Band(models.Model):
    ...
    class Meta: ①
        ordering = [ "name", ] ②
    ...

class Venue(models.Model):
    ...
    class Meta:
        ordering = [ "name", ]
    ...

class Room(models.Model):
    ...
    class Meta:
        ordering = [ "name", ]
    ...
```

① The `Meta` inner class provides additional control information to the Django ORM

② The `ordering` member of `Meta` dictates the default sorting on a queried model; more than one field name can be provided

Exercise 3

Create `admin.ModelAdmin` classes for the `Venue` and `Room` `Model` classes, including custom linked columns methods like with `Musician` and `Band` and searchable name fields.

Class declarations for Django Admin models for venues and rooms are similar to your existing musician and band admin classes. Listing E.20 shows the `VenueAdmin` and `RoomAdmin` class declarations.

Listing E. 20. Django Admin classes for the `Venue` and `Room` classes

```
# RiffMates/bands/admin.py
...
@admin.register(Venue)
class VenueAdmin(admin.ModelAdmin):
    list_display = ("id", "name", "show_rooms")
    search_fields = ("name",)

    def show_rooms(self, obj): ①
        rooms = obj.room_set.all()
        if len(rooms) == 0: ②
            return format_html("<i>None</i>")

        plural = "" ③
        if len(rooms) > 1:
            plural = "s"

        parm = "?id__in=" + ",".join([str(b.id) for b in rooms])
        url = reverse("admin:bands_room_changelist") + parm ④
        return format_html(
            '<a href="{}">Room{}</a>', url, plural) ⑤

    show_rooms.short_description = "Rooms" ⑥


@admin.register(Room)
class RoomAdmin(admin.ModelAdmin):
    list_display = (
        "id",
        "name",
    )
    search_fields = ("name",)
```

① The `.show_rooms()` method is a custom column that displays a link to rooms associated with each `Venue` object.

② If there are no rooms for this venue, the column shows *None*

③ If there is more than one room for this venue, append an "s" to the text showing how many rooms there are

④ The `reverse()` function looks up a URL route; the result can have query parameters to filter the results displayed by the Django Admin

⑤ Use the `format_html()` function to create a link that Django displays without escaping its HTML

⑥ Setting the `.short_description` member of the function changes the title of the associated column

Exercise 4

Add a "Members" column to the `BandAdmin` listing page that contains multiple links, one to

each `Musician` in the `Band`. Note that appending to a string makes it unsafe again. Use `mark_safe` to fix that.

To show the members associated with a band, you need to create and display a custom column that contains a link to the `MembersAdmin` page for each associated musician. The new version of `BandAdmin` is shown in listing E.21.

Listing E. 21. A `BandAdmin` class with a custom column with links to associated musicians

```
# RiffMates/bands/admin.py
...
@admin.register(Band)
class BandAdmin(admin.ModelAdmin):
    list_display = ("id", "name", "show_members")
    search_fields = ("name",)

    def show_members(self, obj): ①
        members = obj.musicians.all() ②
        links = [] ③

        url = reverse("admin:bands_musician_changelist") ④

        for member in members:
            parm = f"?id={member.id}"
            link = format_html(
                '<a href="{}{}">{}</a>', url, parm, member.last_name
            ) ⑤
            links.append(link)

        return mark_safe(", ".join(links)) ⑥

    show_members.short_description = "Members"
```

① A custom column to display links to musicians

② Query all `Musician` objects associated with the `Band` being processed

③ Each musician results in its own link which will be stored in this list

④ Look up the `MusicianAdmin` page's URL

⑤ Use `format_html()` to create a link to each musician's Admin page

⑥ Join the link data into a single string. Processing the strings marks them as unsafe again, so they need to be passed to `mark_safe()` so the HTML is displayed unescaped by the Django Admin

5.5. Chapter 6

Chapter 6 introduced users and user management, including how to store data associated with a user and deal with passwords.

Exercise 1

The `@login_required` decorator only checks if a user got authenticated. A more generic view restriction can be performed using the `@user_passes_test` decorator in the same library. It takes one required argument: a reference to a function that takes the `User` object as an argument and returns `True` if the user should be allowed in. Write a new view `venues_restricted()` that is only visible by those users associated with a `Venue` object and displays a list of all venues. Hint: you can take advantage of other views in your code; they are just functions.

To implement `venues_restricted()` you need two new functions: one for the view itself, and one to pass the `@user_passes_test` decorator. As the view does the same thing as the `venues()` view, it can call it directly, saving you some code. This is a little nonsensical, as you wouldn't provide both a restricted and non-restricted version of the same thing—your users could just use the non-restricted version—but the exercise does demonstrate how to use `@user_passes_test` to implement more restrictions than a required login. Listing E.22 shows both the new view and the function passed to the decorator.

Listing E. 22. Control a view by implementing a restricted function for `@user_passes_test`

```
# RiffMates/bands/views.py
from django.contrib.auth.decorators import user_passes_test
from django.shortcuts import render
...

def has_venue(user): ①
    try:
        return user.userprofile.venues_controlled.count() > 0 ②
    except AttributeError:
        return False ③


@user_passes_test(has_venue) ④
def venues_restricted(request):
    return venues(request) ⑤
```

① This function is passed as a reference to `@user_passes_test`; it should return `True` to indicate the user is allowed in, or `False` otherwise

② Check how many venues the user controls; if it is one or more, allow them in

③ Anonymous users have no profile, which triggers an `AttributeError`: users that are not logged in cannot control a venue, so they fail the test

④ This decorator restricts access to the view based on the result from the function its argument references

⑤ Views are functions; you can use one view to get information from another like any other function

For this view to work it needs a route; for brevity this has been left out. See the exercise in listing E.2 for an example.

Exercise 2

The auth system emits signals on login events. Capture the `django.contrib.auth.signals.user_login_failed` signal and print a message displaying the username and path of the login attempt. See https://docs.djangoproject.com/en/dev/ref/contrib/auth/#topics-auth-signals for details on the arguments passed when receiving this signal. (Note that in production you should use logging rather than `print()`, see appendix B for more details.)

In chapter 6 you learned how to use a signal to activate side-effects when a `User` object is created. Django also provides signals for other things, including login failure. The principle is similar; you use a function decorated with `@receiver` to register it to receive the signal and you can access information about the event through the function's keyword arguments. Listing E.23 shows a function that traps the login-failed signal and prints a message.

Listing E. 23. Invoke a side-effect when a login failure occurs

```
# RiffMates/bands/models.py ①
from django.contrib.auth.signals import user_login_failed ②
from django.dispatch import receiver
...

@receiver(user_login_failed) ③
def track_login_failure(sender, **kwargs):
    username = kwargs["credentials"]["username"] ④
    url = kwargs["request"].path

    print(f"LOGIN Failure by {username} for {url}")
```

① The best place for this signal handler isn't obvious; I've arbitrarily chosen to put it with the other signal handler in the code

② The `user_login_failed` signal is invoked when a username/password mismatch occurs

③ A function registers itself to handle a signal with the `@receiver` decorator, which requires a reference to the signal object

④ The `user_login_failed` signal includes credential information and the request object in its keyword argument dictionary

5.6. Chapter 7

The web is a much more interesting and dynamic place when you handle input from your users. Chapter 7 showed you how to process forms and deal with uploaded files.

Exercise 1

Modify the `musician` view to display an optional description and bio-pic. Add the ability to edit an `edit_musician()` view so users can self-serve the creation and editing of their musician profiles. Insert an "Add Musician Profile" link to the home page, and an "Edit" link to the musician details page if the logged-in user is the owner.

Like with the addition of a description and picture to the `Venue` class in chapter 7, this first requires new fields on the `Musician` class, shown in listing E.24.

Listing E. 24. New fields for the musician model

```
# RiffMates/bands/models.py
...
class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth = models.DateField()
    description = models.TextField(blank=True) ①
    picture = models.ImageField(blank=True, null=True) ②

    ...
```

① A new description field for the musician; uses `blank=True` so its content can be an empty string

② To store a picture you use an `ImageField`, as the field is optional, it can be both `blank` and NULL

With the model in place, a new view is required so the user can edit it. This is shown in listing E.25.

Listing E. 25. A view for editing a `Musician` object

```
# RiffMates/bands/views.py
...
def edit_musician(request, musician_id=0):①
    if musician_id != 0:②
        musician = get_object_or_404(Musician, id=musician_id)
        if (
            not request.user.is_staff
            and not request.user.userprofile.musician_profiles.filter(
                id=musician_id
            ).exists()
        ):③
            raise Http404("Can only edit controlled musicians")

    if request.method == "GET":④
        if musician_id == 0:⑤
            form = MusicianForm()
        else:
            form = MusicianForm(instance=musician)
```

```
    else:  # POST ⑥
        if musician_id == 0:⑦
            musician = Musician.objects.create(birth=date(1900, 1, 1))

        form = MusicianForm(request.POST, request.FILES,
            instance=musician) ⑧

        if form.is_valid(): ⑨
            musician = form.save() ⑩

            # Add the musician to the user's profile if this isn't staff
            if not request.user.is_staff: ⑪
                request.user.userprofile.musician_profiles.add(musician)

            return redirect("musicians") ⑫

    # Was a GET, or Form was not valid ⑬
    data = {
        "form": form,
    }

    return render(request, "edit_musician.html", data)
```

① The view takes a musician's ID as an argument, using `0` to indicate a new musician is to be created

② For a non-zero ID, fetch the corresponding `Musician` object

③ Ensure that only a staff member or someone associated with the `Musician` object can edit it

④ GET is called to present the form

⑤ For new objects, use an empty for; for editing an existing object, pre-populate the form with the musician's info

⑥ POST is called when the form is submitted

⑦ For a new entry, start with a default `Musician` object

⑧ Populate the form with the contents of the POST, any uploaded files, and the existing `Musician` object

⑨ Validate the form's contents

⑩ Saving the form writes the `Musician` object

⑪ If the user is *not* staff, associate their profile with the `Musician` object; this can be called multiple times while creating a single connection

⑫ After a successful edit, send the user to the musician listing page

⑬ For a GET or a failed POST, present the form to the user; the context data contains the form object, which, for a failed POST, contains the previously submitted data

Listing E.26 shows the code adding an "Add Musician" link to the home page.

Listing E. 26. A link for the newly created view

```
<!-- RiffMates/templates/home.html -->
{% extends "base.html" %}
{% load static %}

{% block content %}
  <img src="{% static 'img/logo.png' %}" height="100">
  <h1>Welcome to RiffMates!</h1>

  <ul>
    <li> <a href="{% url 'news' %}">News</a> </li>
    <li> <a href="{% url 'musicians' %}">Musicians</a> </li>
    <li> <a href="{% url 'bands' %}">Bands</a> </li>
    <li> <a href="{% url 'venues' %}">Venues</a> </li>
    <li> <a href="{% url 'list_ads' %}">Classified Ads</a> </li>
    <li> <a href="{% url 'comment' %}">Leave a comment</a> </li>
    <li> <a href="{% url 'add_musician' %}">Add Musician Profile</a> </li> ①

  </ul>

{% endblock content %}
```

① A link to the add/edit view

For an edit link to be added to the Musician Details page, the view needs to be updated to encapsulate the permission. This is similar to how you checked permissions on the `Venue` object. Listing E.27 shows the new version of the `musician()` view.

Listing E. 27. Adding permission information to the `Musician` object in the view

```
# RiffMates/bands/views.py
...
def musician(request, musician_id):
    musician = get_object_or_404(Musician, id=musician_id)
    musician.controller = False ①
    if request.user.is_staff:②
        musician.controller = True
    elif hasattr(request.user, "userprofile"):③
        # This view can be used by people who aren't logged in,
        # those users don't have a profile, need to validate
        # one exists before using it
        profile = request.user.userprofile
        musician.controller = profile.musician_profiles.filter(
            id=musician_id
        ).exists()④

    data = {
        "musician": musician,
    }
```

```
    return render(request, "musician.html", data)
```

① By default, the current user does not have permissions to edit the `Musician` object viewed on the page

② If the logged-in user is staff, they have edit permissions

③ The user doesn't have to be logged in and anonymous users have no profile (see the "Bug alert" in chapter 7)

④ The user can edit the `Musician` object if the object exists in the user's `.musician_profiles` query set

An edit link on the musician details page should only show if the user has edit permissions for that musician; it does this by using the newly added `.controller` member to the `Musician` object in the view. Listing E.28 shows how to do this.

Listing E. 28. Adding the "Edit Musician" link to the musician details page

```
<!-- RiffMates/templates/musician.html -->
{% extends "base.html" %}

{% block title %}
  {{block.super}}: Musician Details
{% endblock %}

{% block content %}
  <h1>{{musician.first_name}} {{musician.last_name}}</h1>

  {% if musician.controller %} ①
    <a href="{% url 'edit_musician' musician.id %}">Edit</a>
  {% endif %}

  {% if musician.picture %} ②
    <p>
      <img src="{{musician.picture.url}}"/>
    </p>
  {% endif %}

  {% if musician.description %} ③
    <p> {{musician.description}} </p>
  {% endif %}

  <p> Was born {{musician.birth}}. </p>
{% endblock content %}
```

① Show an edit link if the user has "controller" permissions

② If there is a bio-pic, display it

③ If there is description content, display it

Exercise 2

Modify the `seeking_ad()` view so that staff and administrators are allowed to edit an ad. Don't forget to change list_ads.html so the edit links show up as well.

Adding staff editing permission to the `seeking_ad()` view requires you to check for staff in both the GET and POST conditions. Listing E.29 shows the new view.

Listing E. 29. Adding permission capabilities for staff to the `seeking_ad()` view

```
# RiffMates/content/views.py
...
@login_required
def seeking_ad(request, ad_id=0):
    if request.method == "GET":
        if ad_id == 0:
            form = SeekingAdForm()
        elif request.user.is_staff:
            ad = get_object_or_404(SeekingAd, id=ad_id)①
            form = SeekingAdForm(instance=ad)
        else:
            ad = get_object_or_404(SeekingAd, id=ad_id, owner=request.user)
            form = SeekingAdForm(instance=ad)

    else:  # POST
        if ad_id == 0:
            form = SeekingAdForm(request.POST)
        elif request.user.is_staff:
            ad = get_object_or_404(SeekingAd, id=ad_id)①
            form = SeekingAdForm(request.POST, instance=ad)
        else:
            ad = get_object_or_404(SeekingAd, id=ad_id, owner=request.user)
            form = SeekingAdForm(request.POST, instance=ad)

        if form.is_valid():
            ad = form.save(commit=False)
            ad.owner = request.user
            ad.save()

            return redirect("list_ads")

    # Was a GET, or Form was not valid
    data = {
        "form": form,
    }

    return render(request, "seeking_ad.html", data)
```

① If the user is staff, the object can be fetched even if the user doesn't own the ad

Inside the corresponding HTML for the view, the check to see if the ad is owned by the logged in user must be modified to also check if they are staff. Listing E.30 shows the small changes necessary.

Listing E. 30. Showing the edit links for staff in the Ad Listing page

```
<!-- RiffMates/templates/list_ad.html -->
<!-- ... -->

    <!-- Inside the Musicians Seeking Bands loop -->
      {% if ad.owner == request.user or request.user.is_staff %} ①
        <a href="{% url 'edit_seeking_ad' ad.id %}">Edit</a>
      {% endif %}

<!-- ... -->
    <!-- Inside the Bands Seeking Musicians loop -->
      {% if ad.owner == request.user or request.user.is_staff %} ①
        <a href="{% url 'edit_seeking_ad' ad.id %}">Edit</a>
      {% endif %}
```

① Check whether the user is the ad's owner or a member of staff

5.7. Chapter 8

Automated testing is extremely important for projects other than the smallest ones. Knowing that your changes didn't break your code makes your project much more stable. The unit testing tools built into Django were covered in chapter 8.

Exercise 1

In chapter 7's exercises you added a view called `edit_musician()` where users could create and manage their own musician profile page. Add a test for that view. Include testing for fetching the form, submitting the form, and adding a musician with a picture. A user can edit the Musician profile they created, as can staff and superusers. Test that those types of users can edit and that a non-owner cannot. Note that you can create a superuser with the `User.objects.create_superuser()` method.

To properly test the `edit_musician()` view, you need to create an admin to validate that they can edit a musician and you will need an image for the musician's bio-pic. Listing E.31 shows the addition of this user and image to the test's set-up.

Listing E. 31. Add a method to test the `edit_musician()` view

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    def setUp(self):
        ...
```

```
        self.PASSWORD = "notsecure"
        ...
        self.admin = User.objects.create_superuser(
            "admin", password=self.PASSWORD
        ) ①

        # Base64 encoded version of a single pixel GIF image
        image = "R0lGODdhAQABAIABAP///wAAACwAAAAAAQABAAACAkQBADs="
        self.image = b64decode(image) ②
```

① Add an administrator account and keep a reference in the test object for future use

② This Base64-encoded GIF image can be used instead of loading an image from a file

Testing `edit_musician()` is done in steps; listing E.32 validates the creation of a new musician.

Listing E. 32. Test the creation of a new musician

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    ...
    def test_edit_musician(self):
        self.client.login(username="owner",
            password=self.PASSWORD) ①

        # Verify the page fetch works
        url = "/bands/edit_musician/0/"
        response = self.client.get(url) ②
        self.assertEqual(200, response.status_code) ③

        # Create a new Musician (NB: one already exists from .setup())
        file = SimpleUploadedFile("test.gif", self.image) ④
        data = {
            "first_name": "First2",
            "last_name": "Last2",
            "birth": date(1900, 1, 2),
            "description": "Description2",
            "picture": file,
        }
        response = self.client.post(url, data) ⑤

        self.assertEqual(302, response.status_code) ⑥

        # Validate the Musician was created ⑦
        musician = Musician.objects.last()
        self.assertEqual(data["first_name"], musician.first_name)
        self.assertEqual(data["last_name"], musician.last_name)
        self.assertEqual(data["description"], musician.description)
        self.assertEqual(data["birth"], musician.birth)
        self.assertIsNotNone(musician.picture)
```

```
        self.assertTrue(
            self.owner.userprofile.musician_profiles.filter(
                id=musician.id
            ).exists()
        )
```

① Start by logging in as a user that will own the new `Musician` object

② Fetch the page with the form

③ Validate that viewing the page returned a 200 ("ok") response code

④ Create an uploaded file object to contain the musician's bio-pic

⑤ POST the form data, including the image file, to the page

⑥ A successful form submission redirects (response code 302) the user to a new page

⑦ Load the newly created `Musician` object and validate that the fields match what was sent in the form

Once creation of the object is verified, you can check that object editing works. Listing E.33 validates this test scenario.

Listing E. 33. Validate object editing

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    ...
    def test_edit_musician(self):
        ...
        # Now edit the Musician (recreate the file to reset its stream)
        url = f"/bands/edit_musician/{musician.id}/" ①
        data["first_name"] = "Edited Name"
        file = SimpleUploadedFile("test.gif", self.image) ②
        data["picture"] = file
        response = self.client.post(url, data) ③

        self.assertEqual(302, response.status_code) ④
        musician = Musician.objects.last() ⑤
        self.assertEqual(data["first_name"], musician.first_name) ⑥
```

① The URL for editing a `Musician` object uses the object's ID

② The file stream was consumed earlier in the test; you need a new uploaded file object to test with

③ Post the changed field

④ Verify a successful redirect

⑤ Fetch the most recent object

⑥ Validate that the name was changed per the submitted form

Finally, you should test that an admin can perform an edit and non-owners cannot. Listing E.34 verifies this scenario.

Listing E. 34. Validate admin editing and non-owner edits are rejected

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    ...
    def test_edit_musician(self):
        ...
        # Verify that a superuser can edit
        self.client.login(username="admin",
            password=self.PASSWORD) ①
        file = SimpleUploadedFile("test.gif", self.image)
        data["picture"] = file
        data["first_name"] = "Super Edited Name"
        response = self.client.post(url, data) ②

        self.assertEqual(302, response.status_code) ③
        musician = Musician.objects.last()
        self.assertEqual(data["first_name"], musician.first_name)

        # Verify that a non-owner can't edit
        self.client.login(username="member",
            password=self.PASSWORD) ④
        response = self.client.post(url, data)
        self.assertEqual(404, response.status_code) ⑤
```

① Log in as the admin account instead

② Perform the edit actions like before, but as an admin

③ Validate the correct redirect response and that the edit took place

④ Login as a non-owner

⑤ Validate that a 404 response was returned for a user without edit permissions

Exercise 2

Write a test for the `musicians()` view that validates that pagination is working. The view needs to be called multiple times with a variety of query parameters to test different settings for the number of items per page as well as the next and previous arguments. Note that the paginator's `.has_previous()` and `.has_next()` are methods, not attributes: you didn't need the parentheses in the template but you do when calling them in code.

To verify pagination is working you need more data, so the first part of the test is to create more musicians. I have chosen to create nine more, giving ten in total. Listing E.35 shows this.

Listing E. 35. Create more test data for pagination

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    ...
    def test_musicians(self):
        # Create 9 more Musicians (one exists from .setUp())
        for x in range(2, 11):
            Musician.objects.create(
                first_name=f"First{x}",
                last_name=f"Last{x}",
                birth=date(1900, 1, x),
            )
```

With the new musician data in place, start by fetching all of them on a single page. The view defaults to ten per page, so no query parameters should result in no pagination. Listing E.36 validates this scenario.

Listing E. 36. No query parameters and no pagination

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    ...
    def test_musicians(self):
        ...
        # Test URL with no arguments, 10 musicians means no pagination
        url = "/bands/musicians/" ①
        response = self.client.get(url)

        self.assertEqual(200, response.status_code) ②
        self.assertEqual(10, len(response.context["musicians"])) ③
        self.assertFalse(response.context["page"].has_previous()) ④
        self.assertFalse(response.context["page"].has_next()) ⑤
```

① No query parameters in the URL

② Always verify that 200 ("ok") came back

③ Check there are ten musicians in the context

④ The `.has_previous()` call should return `False`

⑤ The `.has_next()` call should return `False`

Next, it is time to test using a query parameter to change the number of items on the page and that pagination works. Listing E.37 sets the items per page to five, resulting in two pages.

Listing E. 37. Test five musicians per page

```
# RiffMates/bands/tests.py
```

```
...
class TestBands(TestCase):
    ...
    def test_musicians(self):
        ...
        # Test with 5 per page
        url = "/bands/musicians/?items_per_page=5" ①
        response = self.client.get(url)

        self.assertEqual(200, response.status_code)
        self.assertEqual(5, len(response.context["musicians"])) ②
        self.assertFalse(response.context["page"].has_previous())
        self.assertTrue(response.context["page"].has_next())

        # Test with 5 per page, page #2
        url = "/bands/musicians/?items_per_page=5&page=2" ③
        response = self.client.get(url)

        self.assertEqual(200, response.status_code)
        self.assertEqual(5, len(response.context["musicians"])) ④
        self.assertTrue(response.context["page"].has_previous())
        self.assertFalse(response.context["page"].has_next())
```

① Use the `items_per_page` query parameter to set five musicians per page

② Validate that the right number of musicians came back and that there is *no* previous page, but there *is* a next page

③ Set both the number of items on the page as well as the page number

④ Validate the right number of musicians came back and there *is* a previous page but *no* next page

Good tests check for error conditions as well. Listing E.38 checks that a bad value for the number of items per page is handled properly.

Listing E. 38. Handle bad query parameter values as well

```
# RiffMates/bands/tests.py
...
class TestBands(TestCase):
    ...
    def test_musicians(self):
        ...
        # Test bad page size
        url = "/bands/musicians/?items_per_page=-1" ①
        response = self.client.get(url)

        self.assertEqual(200, response.status_code)
        self.assertEqual(10, len(response.context["musicians"])) ②
        self.assertFalse(response.context["page"].has_other_pages())
```

① Negative numbers for `items_per_page` are not allowed

② Bad values result in the default of ten musicians per page

5.8. Chapter 9

Sometimes a web-based interface isn't the best choice, especially for dealing with managerial tasks for your site. Chapter 9 covers how to write your own Django management command.

Exercise 1

Create a `venues` management command that lists all the Venues in your database. Include a `--rooms` flag to optionally display the names of the rooms associated with a venue. Hint: the `action="store_true"` parameter to `add_arguments` changes your flag so it acts as a boolean flag and doesn't expect any additional arguments

Listing E.39 shows the management command to list venues and optionally list their associated rooms.

Listing E. 39. Management command to list venues

```
# RiffMates/bands/management/commands/venues.py
from bands.models import Venue
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    help = "Lists registered venues"

    def add_arguments(self, parser): ①
        parser.add_argument( ②
            "--rooms",
            "-r",
            action="store_true", ③
            help="Display a venue's room information",
        )

    def handle(self, *args, **options):
        for venue in Venue.objects.all(): ④
            self.stdout.write(venue.name) ⑤

            if options["rooms"]: ⑥
                for room in venue.room_set.all(): ⑦
                    self.stdout.write("    " + room.name)
```

① Use the `.add_arguments()` method to configure command-line arguments for your management command

② The `.add_argument()` method on the parser registers a new command-line argument

③ Using `action="store_true"` indicates that this flag acts as a boolean and doesn't require any arguments to go with it

④ Iterate through all venues

⑤ Remember to use `self.stdout.write()` instead of `print()`

⑥ If the user used `--rooms`, the `options["rooms"]` value will be `True`

⑦ Iterate over the rooms associated with this venue

Exercise 2

When you delete an object with a `FileField` or `ImageField`, Django does not remove the associated file. Write a `cleanup` command that searches through all the Musician and Venue objects for their associated pictures and compares this to the uploaded files, removing any orphans. Hint 1: use a Python `set` to contain all picture references and another to contain all file references, then use the `set.difference()` method to determine the orphan list. Hint 2: convert the picture values into `pathlib.Path` objects before comparing them. Hint 3: test your code a lot, printing out results instead of deleting files until you're absolutely sure it works.

As the `cleanup` command can be destructive (it removes files), providing a `--show` option that only shows what would be done is useful for the user. Listing E.40 shows the setup and argument configuration for the command.

Listing E. 40. Configuring arguments for the `cleanup` command

```
# RiffMates/bands/management/commands/cleanup.py
from pathlib import Path

from bands.models import Musician, Venue
from django.conf import settings
from django.core.management.base import BaseCommand, CommandError

class Command(BaseCommand):
    help = "Removes uploaded files not owned by a Musician or Venue"

    def add_arguments(self, parser):
        parser.add_argument(
            "--show",
            "-s",
            action="store_true",
            help=("Show which files would be removed but don't remove them"),
        ) ①
```

① Add a boolean `--show` flag to the command

To determine what files should be removed, you need to create a set containing all the files associated with `Musician` and `Venue` objects and find the difference between that and the files on disk. Listing E.41 finds the orphaned file set.

Listing E. 41. Determine what files should be removed

```
# RiffMates/bands/management/commands/cleanup.py
```

```
...

class Command(BaseCommand):
    ...
    def handle(self, *args, **options):
        model_set = set() ①
        for musician in Musician.objects.all(): ②
            if musician.picture:
                model_set.add(Path(musician.picture.path))

        for venue in Venue.objects.all(): ③
            if venue.picture:
                model_set.add(Path(venue.picture.path))

        file_set = set(settings.MEDIA_ROOT.glob("**/*")) ④

        orphaned = file_set.difference(model_set) ⑤
```

① Store the files used by models as a `set`

② Iterate over all `Musician` objects and add each bio-pic to the set

③ Iterate over all `Venue` objects and add each picture to the set

④ Find all the files on the hard-drive

⑤ The difference between the `model_set` and the `file_set` subtracts all of the model and venue files from the whole list, leaving just those which are orphaned

Once the set of orphans is determined you can perform the removal action on it. Listing E.42 conservatively comments out the actual deletion. Only uncomment the `path.unlink()` line if you're sure your code works.

Listing E. 42. Determine what files should be removed

```
# RiffMates/bands/management/commands/cleanup.py
...

class Command(BaseCommand):
    ...
    def handle(self, *args, **options):
        ...
        if not orphaned: ①
            self.stdout.write("No orphaned files")
            exit()

        if options["show"]: ②
            self.stdout.write(f"Orphaned files: ({len(orphaned)})")
            for path in orphaned:
                if path.is_file():
                    self.stdout.write("    " + str(path))
        else: ③
            self.stdout.write(f"Removing: ({len(orphaned)})")
```

```
            for path in orphaned:
                if path.is_file():
                    self.stdout.write("    " + str(path))

                    # Uncomment the following to do actual damage
                    # path.unlink() ④
```

① If the set of orphans was empty, inform the user there is nothing to delete

② If the `--show` flag is present, only display the orphaned files, don't delete them

③ If there are orphans and the `--show` flag is *not* present, it is time to remove files

④ This code iterates over the orphaned files and deletes them if this line is uncommented

5.9. Chapter 10

Django ORM `Model` objects are only proxies to the data in the actual database. As you change your models, the database has to change as well. Chapter 10 is about dealing with migration files, which control the corresponding table changes.

Exercise 1

Modify the `Promoter` model by adding a "street address" field, migrating the change. Then modify it again, adding fields for city, province/state, country, and postal/zip code. Create a migration script, migrate it, then squash the migrations together.

First, add the "street address" field to the `Promoter` model as in listing E.43.

Listing E. 43. Add a street address field to the `Promoter` model

```
# RiffMates/promoters/models.py
from django.db import models

class Promoter(models.Model):
    common_name = models.CharField(max_length=25)
    full_name = models.CharField(max_length=50)
    famous_for = models.CharField(max_length=50)
    birth = models.DateField(blank=True, null=True)
    death = models.DateField(blank=True, null=True)
    street_address = models.CharField(blank=True,
        default="", max_length=25) ①
```

① The new field for a street address

Creating a migration for this change adds 0008_promoter_street_address.py to your RiffMates/promoters/migrations/ directory.

Next, modify the same class, adding fields for city, country, postal/zip code, and province/state, as shown in listing E.44.

Listing E. 44. Add additional fields to the `Promoter` object

```
# RiffMates/promoters/models.py
from django.db import models

class Promoter(models.Model):
    common_name = models.CharField(max_length=25)
    full_name = models.CharField(max_length=50)
    famous_for = models.CharField(max_length=50)
    birth = models.DateField(blank=True, null=True)
    death = models.DateField(blank=True, null=True)
    street_address = models.CharField(blank=True, default="", max_length=25)
    city = models.CharField(blank=True, default="",
        max_length=25) ①
    country_code = models.CharField(blank=True, default="", max_length=2)
    postal_zip = models.CharField(blank=True, default="", max_length=12)
    province_state = models.CharField(blank=True, default="", max_length=25)
```

① Four new fields for city, country, postal/zip, and province/state

Creating a migration for this change adds the awkwardly named 0009_promoter_country_code_promoter_postal_zip_and_more.py file to your collection.

Running the `squash` command creates the even more awkwardly named: 0008_promoter_street_address_squashed_0009_promoter_country_code_promoter_postal_zip_and_more.py. If you examine the resulting migration file you will find all the added fields from the two migration steps in this single file.

Exercise 2

Convert the address fields in exercise 1 into a single field called "address", a single multi-line text field containing all the address information. Write migration scripts for adding the new field and converting any data from the old fields to the new ones. Finally, finish the process by removing the old fields and migrating that change as well.

Before doing anything, put some address data in your database so you can test your migration command. To merge the fields added in exercise 1 into a single `TextField`, you need to add the destination field. Listing E.45 shows the new state of `Promoter`.

Listing E. 45. The `Promoter` object with the new destination combined field

```
# RiffMates/promoters/models.py
from django.db import models

class Promoter(models.Model):
    common_name = models.CharField(max_length=25)
    full_name = models.CharField(max_length=50)
    famous_for = models.CharField(max_length=50)
    birth = models.DateField(blank=True, null=True)
```

```
    death = models.DateField(blank=True, null=True)
    street_address = models.CharField(blank=True, default="", max_length=25)
    city = models.CharField(blank=True, default="", max_length=25)
    country_code = models.CharField(blank=True, default="", max_length=2)
    postal_zip = models.CharField(blank=True, default="", max_length=12)
    province_state = models.CharField(blank=True, default="", max_length=25)
    address = models.TextField(blank=True, default="") ①
```

① The new "combined" address field

Run `makemigrations` to create 0010_promoter_address.py. Then run it again using the `--empty promoters` option to create a new migration file for your data operations. I renamed mine to 0011_data_address.py, which is shown in listing E.46 including the added data operation.

Listing E. 46. A migration script with a data operation in it

```
# RiffMates/promoters/migrations/0011_data_address.py
from django.db import migrations

def group_address(apps, schema_editor): ①
    Promoter = apps.get_model("promoters", "Promoter") ②
    for promoter in Promoter.objects.all(): ③
        text = "" ④
        if promoter.street_address:
            text += promoter.street_address + "\n"
        if promoter.city:
            text += promoter.city + "\n"
        if promoter.province_state:
            text += promoter.province_state + "\n"
        if promoter.country_code:
            text += promoter.country_code + "\n"
        if promoter.postal_zip:
            text += promoter.postal_zip + "\n"

        promoter.address = text ⑤
        promoter.save()

class Migration(migrations.Migration):
    dependencies = [ ⑥
        (
            "promoters",
            "0010_promoter_address",
        ),
    ]

    operations = [
        # Can't undo this migration, there is no way of which lines were
        # missing in the grouping
        migrations.RunPython(group_address), ⑦
    ]
```

① This function is called by `migrations.RunPython` to perform data operations

② The `apps.get_model()` function gets the version of the `Promoter` model that is correct at this step in the migration process

③ Iterate over all promoters

④ Create a string based on the combined values of the address fields

⑤ Copy the string into the new field and save the model

⑥ Dependencies are created automatically from the `makemigrations` command

⑦ As there is no way to undo this data operation, the `RunPython` command only takes a single reference: the `group_address` function

Once your data migration is complete, you can modify `Promoter` to remove the now extra address fields. Running `makemigrations` on that will result in the new file: 0012_remove_promoter_city_remove_promoter_country_code_and_more.py.

5.10. Chapter 11

Adding an API to your project enables you to provide access to your data to other systems, whether those are consumers of your data or alternate user interfaces such as mobile apps or Single Page Applications.

Exercise 1

Write an API to list all the `Band` objects in the system with the corresponding `Musician` data nested within each `Band` object.

To build a band listing API you need schemas for the `Band` and `Musician` objects. These can both be sub-classes of `ModelSchema` to minimize the amount of code required. Lisiting E.47 shows the required code.

Listing E. 47. Serializers for the `Band` and `Musician` objects

```
# RiffMates/bands/api.py
...
from ninja import ModelSchema
...
from bands.models import Band, Musician
...
class MusicianOut(ModelSchema): ①
    class Config: ②
        model = Musician ③
        model_fields = ["id", "first_name", "last_name", "birth",
            "description",] ④


class BandSchema(ModelSchema):
    musicians: list[MusicianOut] ⑤
```

```
    class Config:
        model = Band
        model_fields = ["id", "name"]
```

① Sub-class `ModelSchema` to quickly create a serializer for the `Musician` class

② The inner `Config` class gets used to define the serializer's association with the ORM class

③ The `model` field defines which ORM model is associated with this schema

④ The `model_fields` value contains the list of fields from `Musician` that participate in this serializer

⑤ To nest musicians in a band serializer, define a field that relates to the `Musician` schema

With the appropriate serializers in place, the API endpoint is a simple view that returns a list of all the `Band` objects. Listing E.48 shows the code.

Listing E. 48. A migration script with a data operation in it

```
# RiffMates/bands/api.py
...
router = Router() ①
...
@router.get("/bands/", response=list[BandSchema]) ②
def list_bands(request):
    return Band.objects.all() ③
```

① API endpoints are registered against the single `Router` defined in the file, created earlier in chapter 11

② A band-listing endpoint responds with a list of `BandSchema` serializations

③ You return a `QuerySet` of `Band` objects to Ninja and it takes care of the rest

Exercise 2

Write an API to update the fields of a `Musician` object, requiring authentication to do so.

The `MusicianOut` serializer defined in the previous exercise includes the `Musician` ID. This ID should not be included in the set of fields used to update an object, as the ID is provided as part of the URL. This means you'll need a different serializer for this end-point. Listing [che-listing-11-2-schema] shows this variation.

Listing E. 49. Serializers for the `Band` and `Musician` objects

```
# RiffMates/bands/api.py
...
from ninja import ModelSchema
...
from bands.models import Musician
...
```

```
class MusicianIn(ModelSchema):
    class Config:
        model = Musician
        model_fields = ["first_name", "last_name", "birth", "description",]
```

The API endpoint view uses the PUT action to do an update requiring all field information be set. The response of the PUT call is the modified `Musician` object. This endpoint requires authentication, which in this case simply means specifying the `APIKeyHeader` object used throughout chapter 11.

Inside the view, the code loops through each of the fields in the serializer, applying the new value to the `Musician` object. Once complete, the object gets saved and returned. Listing E.50 shows the code required.

Listing E. 50. Serializers for the `Band` and `Musician` objects

```
# RiffMates/bands/api.py
...
from api_auth import api_key ①
...
@router.put(
    "/musician/{musician_id}/",
    response=MusicianOut, ②
    auth=api_key) ③
def update_musician(request, musician_id, payload:MusicianIn): ④
    musician = get_object_or_404(Musician, id=musician_id) ⑤
    for key, value in payload.dict().items(): ⑥
        setattr(musician, key, value)

    musician.save() ⑦
    return musician
```

① Import the `APIKeyHeader` object from the shared module

② The response from a PUT action is a serialization of the modified `Musician` object

③ The `auth` argument takes an authentication object, in this case the shared-key mechanism

④ The view requires the ID of the musician to edit and the payload body containing the serialized `Musician` containing the new field values

⑤ Fetch the `Musician` object for the given ID

⑥ Loop through the payload and for each key/value pair, update the corresponding field in the `Musician` object

⑦ Save the changes to the `Musician` object and return it so Ninja can serialze the new values

5.11. Chapter 12

Django was originally built for the web 1.0 world. To add more interactivity you can use a heavy JavaScript framework like React or a lighter-weight library like HTMX. HTMX provides API-like functionality through HTML attributes, dynamically replacing portions of a page.

Exercise 1

Add search functionality for your ad content similar to the musicians search page. Implement both search-as-you-type and infinite scrolling.

The HTMX `hx-get` attribute interacts with an input box by appending its contents to the URL's query parameter. This can be used to build a search-as-you-type feature. Listing E.51 contains the HTML for an ad-searching page.

Listing E. 51. HTML for a search ads page

```
<!-- RiffMates/templates/search_ads.html -->
{% extends "base.html" %}

{% block content %}
  <h1>Search Ads</h1>

  <input name="search_text"
    placeholder="Search the ads..."
    type="text"
    value="{{search_text}}" ①
    hx-get="{% url 'search_ads' %}" ②
    hx-trigger="keyup changed delay:500ms" ③
    hx-target="#results" ④
    hx-push-url="true"> ⑤

    <div class="results" id="results"> ⑥
      {% include "partials/ad_results.html" %} ⑦
    </div>
{% endblock content %}
```

① To support deep linking, populate the input box with any search term given to the page at render time

② The `hx-get` attribute specifies the URL to call when content is added to the input box

③ The `hx-trigger` attribute dictates when the GET is performed. This value is based on a `keyup` event, but only if it changes the contents of the input box and occurs 500ms after the last press.

④ When new content is fetched, put it in the `<div>` with ID `results`

⑤ Append the contents of the input box to the query parameter in the browser's nav to enable deep linking and bookmarking

⑥ The `<div>` tag that is the target of results

⑦ To support deep linking, include the sub-template that populates any results available at render time

Following the common pattern in chapter 12, the HTML is broken into two parts: the main page and the partial template used to render the results. The partial template displays the ad search results.

To implement infinite scroll on the search results, the search data gets paginated. The partial that displays the results also needs to contain the code that triggers fetching more data. It does this by using a value of `revealed` for the `hx-trigger` attribute, calling for the next page of data to display. Listing E.52 shows the complete HTML for displaying a partial, with both the page of search results as well as the code for triggering the fetch action for next chunk.

Listing E. 52. HTML partial to display an ad

```
<!-- RiffMates/templates/partials/ad_results.html -->
{% for ad in ads %} ①
  <p>
    {{ad.date}} &mdash;
    {{ad.musician.first_name}} {{ad.musician.last_name}}

    <br/>
    {% if ad.seeking == "M" %}
      Band, "{{ad.band.name}}", is seeking a musician
    {% else %}
      Musician, "{{ad.musician.first_name}} {{ad.musician.last_name}}"
      is seeking a band
    {% endif %}
    <br/>
    <i>{{ad.content}}</i>
  </p>
  <hr>
{% empty %}
  <p> <i>No ads at this time</i> </p>
{% endfor %}

{% if has_more %} ②
  <div ③
    hx-get="{% url 'search_ads' %}?page={{next_page}}&search_text={{search_text}}"
    hx-trigger="revealed" ④
    hx-swap="outerHTML"> ⑤
    <i> Loading more results ... </i> ⑥
  </div>
{% endif %}
```

① Loop through the ads given and display them to the user

② The Django paginator indicates if there is more data

③ This tag gets replaced when more results are loaded. The `hx-get` value fetches data from the next page for the given search terms

④ The GET is triggered when this tag is displayed on screen

⑤ A value of `outerHTML` means the entire tag gets replaced rather than just the contents within the tag

⑥ A placeholder to display until the results have returned

The view for performing search is responsible for both displaying the main page and a partial for

result data. The search text is split into terms, and a `Q` object is built to perform the search. The search results are paginated, and the partials can be rendered a "page" at a time. Listing E.53 contains the view code.

Listing E. 53. An HTMX-capable view for searching ads

```
# RiffMates/content/views.py
import urllib
from time import sleep
...

from django.core.paginator import Paginator
from django.db.models import Q
from django.shortcuts import render

from bands.views import _get_items_per_page, _get_page_num ①
...

def search_ads(request):
    search_text = request.GET.get("search_text", "") ②
    search_text = urllib.parse.unquote(search_text)
    search_text = search_text.strip()

    ads = []

    if search_text:
        parts = search_text.split() ③

        q = Q(content__icontains=parts[0]) ④
        for part in parts[1:]:
            q |= Q(content__icontains=part) ⑤

        ads = SeekingAd.objects.filter(q) ⑥

    items_per_page = _get_items_per_page(request) ⑦
    paginator = Paginator(ads, items_per_page)
    page_num = _get_page_num(request, paginator)
    page = paginator.page(page_num)

    data = {
        "search_text": search_text,
        "ads": page.object_list,
        "has_more": page.has_next(),
        "next_page": page_num + 1,
    }

    if request.htmx: ⑧
        if page_num > 1:
            sleep(2)

        return render(request, "partials/ad_results.html", data)
```

```
    return render(request, "search_ads.html", data) ⑨
```

① Import the utilities for handling pagination from bands/views.py. A better design would move these to a common utilities file

② Get the search terms from the query parameter, decoding it and removing extra white-space

③ Split the search string into a series of terms

④ Look for the first search term in the ad's `content` field, case insensitive

⑤ Add each remaining search term to the query object

⑥ Run the query to find the ads

⑦ Paginate the results for "pages" used by the infinite scroll feature

⑧ If this is an HTMX call, return a partial with results. The call to `sleep()` helps you see the results.

⑨ For the non-HTMX case, render the main page

Exercise 2

Create a page that has two tabs: one for musicians and one for venues. You'll need a new view for the page, but the existing listing views for musicians and venues can be used to populate the tabs. There is an example on how to write tabs on the HTMX site: https://htmx.org/examples/tabs-hateoas/

A tabbed interface on a page consists of a link for each tab, where clicking the link performs a GET to fetch the page's contents and dynamically replace it inside the main page. One tricky problem here is that every page currently has a common nav bar. To get around this, the nav in base.html needs to be modified so it is contained in a `{% block %}` tag so it can be over-loaded. Listing E.54 shows the change to RiffMates/templates/base.html.

Listing E. 54. Modify the parent template so the nav bar can be over-loaded

```
<!-- RiffMates/templates/base.html -->
<!-- ... -->
<body>
  {% block nav %} ①
  <div style="float: right;"> ②
    <a href="{% url 'home' %}">Home</a>   
    <!-- ... -->

  </div>
  {% endblock nav %}
  <!-- ... -->
```

① Wrap the nav in a `{% block %}` tag for later

② The current `<div>` for the nav

With the newly wrapped nav, the content for the tabbed page can override it so it only gets

displayed once. Listing E.55 shows the HTML which includes the `hx-get` attributes for fetching each tab's contents.

Listing E. 55. HTML for the tabbed page

```
<!-- RiffMates/templates/tabbed_listing.html -->
{% extends "base.html" %}

{% block nav %} ①
  {# empty block to ensure you don't get the nav twice #}
{% endblock %}

{% block content %}
  <div> ②
    <a
      hx-get="/bands/musicians/" ③
      hx-target="#tab-block" ④
      hx-swap="innerHTML" ⑤
      style="cursor: pointer; text-decoration: underline;" ⑥
      >Musicians</a>
       

    <a
      hx-get="/bands/venues/"
      hx-target="#tab-block"
      hx-swap="innerHTML"
      style="cursor: pointer; text-decoration: underline;"
      >Venues</a>

  </div>

  <div id="tab-block"> ⑦
    <div
        hx-get="/bands/musicians/" ⑧
        hx-trigger="load" ⑨
        hx-swap="outerHTML"/>
    </div>
  </div>
{% endblock content %}
```

① Override the nav block, rendering it as empty; the page's nav will be displayed by the included content page

② A container for the tabbed interface

③ Each tab is a link with the `hx-get` fetching the content from an existing view

④ The target tag to replace with fetched content

⑤ Only replace the contents inside the target tag

⑥ Lack of an `href` attribute means the browser needs extra styling information

⑦ The container for the displayed contents

⑧ The default tab on page load is the first one, fetch it

⑨ Trigger the default tab as son as the page is loaded