3

Extending Your Blog Application

The previous chapter went through the basics of forms and the creation of a comment system. You also learned how to send emails with Django. In this chapter, you will extend your blog application with other popular features used on blogging platforms, such as tagging, recommending similar posts, providing an RSS feed to readers, and allowing them to search posts. You will learn about new components and functionalities with Django by building these functionalities.

The chapter will cover the following topics:

- Integrating third-party applications
- Using `django-taggit` to implement a tagging system
- Building complex QuerySets to recommend similar posts
- Creating custom template tags and filters to show a list of the latest posts and most commented posts in the sidebar
- Creating a sitemap using the sitemap framework
- Building an RSS feed using the syndication framework
- Installing PostgreSQL
- Implementing a full-text search engine with Django and PostgreSQL

The source code for this chapter can be found at `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03`.

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

Adding the tagging functionality

A very common functionality in blogs is to categorize posts using tags. Tags allow you to categorize content in a non-hierarchical manner, using simple keywords. A tag is simply a label or keyword that can be assigned to posts. We will create a tagging system by integrating a third-party Django tagging application into the project.

`django-taggit` is a reusable application that primarily offers you a `Tag` model and a manager to easily add tags to any model. You can take a look at its source code at `https://github.com/jazzband/django-taggit`.

First, you need to install `django-taggit` via `pip` by running the following command:

```
pip install django-taggit==3.0.0
```

Then, open the `settings.py` file of the `mysite` project and add `taggit` to your `INSTALLED_APPS` setting, as follows:

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

Open the `models.py` file of your `blog` application and add the `TaggableManager` manager provided by `django-taggit` to the `Post` model using the following code:

```
from taggit.managers import TaggableManager

class Post(models.Model):
    # ...
    tags = TaggableManager()
```

The `tags` manager will allow you to add, retrieve, and remove tags from `Post` objects.

The following schema shows the data models defined by `django-taggit` to create tags and store related tagged objects:

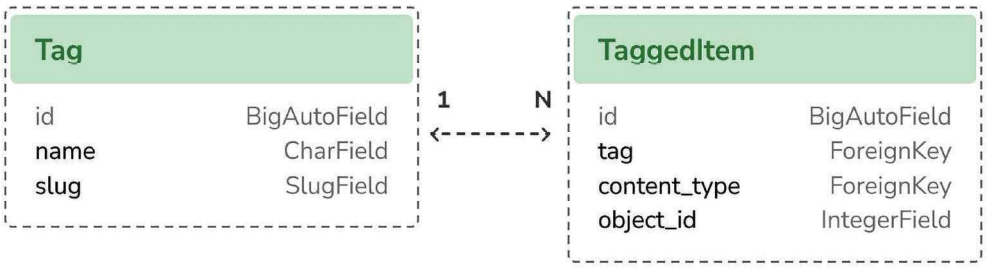

Figure 3.1: Tag models of django-taggit

The `Tag` model is used to store tags. It contains a `name` and a `slug` field.

The `TaggedItem` model is used to store the related tagged objects. It has a `ForeignKey` field for the related `Tag` object. It contains a `ForeignKey` to a `ContentType` object and an `IntegerField` to store the related `id` of the tagged object. The `content_type` and `object_id` fields combined form a generic relationship with any model in your project. This allows you to create relationships between a `Tag` instance and any other model instance of your applications. You will learn about generic relations in *Chapter 7, Tracking User Actions*.

Run the following command in the shell prompt to create a migration for your model changes:

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

You should get the following output:

```
Migrations for 'blog':
  blog/migrations/0004_post_tags.py
    - Add field tags to post
```

Now, run the following command to create the required database tables for `django-taggit` models and to synchronize your model changes:

```
python manage.py migrate
```

You will see an output indicating that migrations have been applied, as follows:

```
Applying taggit.0001_initial... OK
Applying taggit.0002_auto_20150616_2121... OK
Applying taggit.0003_taggeditem_add_unique_index... OK
Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
Applying taggit.0005_auto_20220424_2025... OK
Applying blog.0004_post_tags... OK
```

The database is now in sync with the `taggit` models and we can start using the functionalities of `django-taggit`.

Let's now explore how to use the `tags` manager.

Open the Django shell by running the following command in the system shell prompt:

```
python manage.py shell
```

Run the following code to retrieve one of the posts (the one with the `1` ID):

```
>>> from blog.models import Post
>>> post = Post.objects.get(id=1)
```

Then, add some tags to it and retrieve its tags to check whether they were successfully added:

```
>>> post.tags.add('music', 'jazz', 'django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>, <Tag: django>]>
```

Finally, remove a tag and check the list of tags again:

```
>>> post.tags.remove('django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>]>
```

It's really easy to add, retrieve, or remove tags from a model using the manager we have defined.

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

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/taggit/tag/` in your browser.

You will see the administration page with the list of `Tag` objects of the `taggit` application:

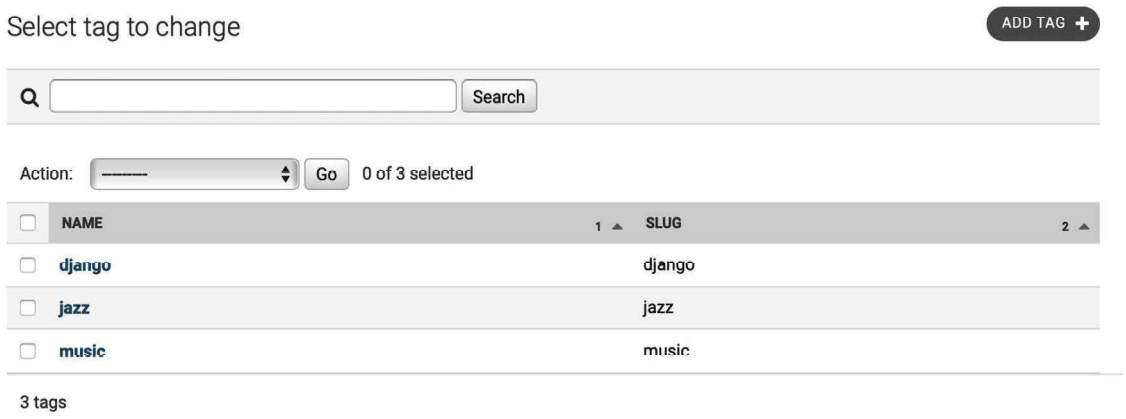

Figure 3.2: The tag change list view on the Django administration site

Click on the **jazz** tag. You will see the following:

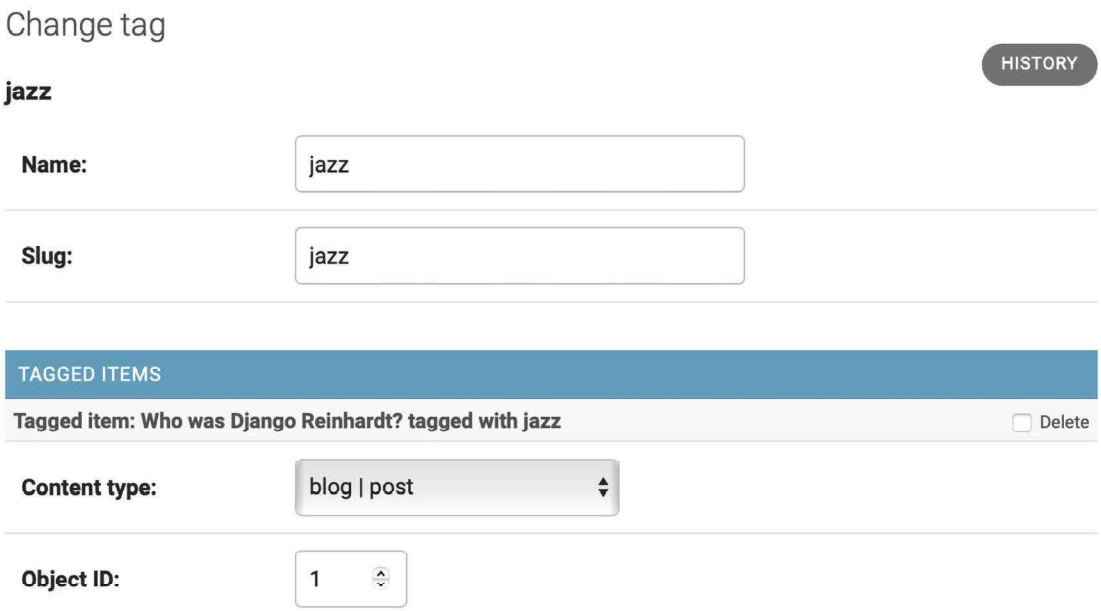

Figure 3.3: The related tags field of a Post object

Navigate to `http://127.0.0.1:8000/admin/blog/post/1/change/` to edit the post with ID `1`.

You will see that posts now include a new **Tags** field, as follows, where you can easily edit tags:

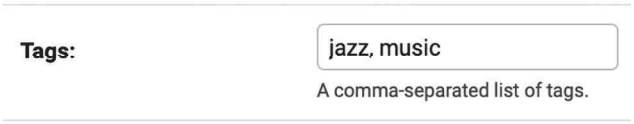

Figure 3.4: The related tags field of a Post object

Now, you need to edit your blog posts to display tags.

Open the `blog/post/list.html` template and add the following HTML code highlighted in bold:

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

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

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=page_obj %}
{% endblock %}
```

The `join` template filter works the same as the Python string `join()` method to concatenate elements with the given string.

Open `http://127.0.0.1:8000/blog/` in your browser. You should be able to see the list of tags under each post title:

Figure 3.5: The Post list item, including related tags

Next, we will edit the `post_list` view to let users list all posts tagged with a specific tag.

Open the `views.py` file of your `blog` application, import the `Tag` model from `django-taggit`, and change the `post_list` view to optionally filter posts by a tag, as follows. New code is highlighted in bold:

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    post_list = Post.published.all()
    tag = None
    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        post_list = post_list.filter(tags__in=[tag])
    # Pagination with 3 posts per page
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # If page_number is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
        # If page_number is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts,
                   'tag': tag})
```

The `post_list` view now works as follows:

1. It takes an optional `tag_slug` parameter that has a `None` default value. This parameter will be passed in the URL.
2. Inside the view, we build the initial QuerySet, retrieving all published posts, and if there is a given tag slug, we get the `Tag` object with the given slug using the `get_object_or_404()` shortcut.
3. Then, we filter the list of posts by the ones that contain the given tag. Since this is a many-to-many relationship, we have to filter posts by tags contained in a given list, which, in this case, contains only one element. We use the `__in` field lookup. Many-to-many relationships occur when multiple objects of a model are associated with multiple objects of another model. In our application, a post can have multiple tags and a tag can be related to multiple posts. You will learn how to create many-to-many relationships in *Chapter 6*, *Sharing Content on Your Website*. You can discover more about many-to-many relationships at `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/`.
4. Finally, the `render()` function now passes the new `tag` variable to the template.

Remember that QuerySets are lazy. The QuerySets to retrieve posts will only be evaluated when you loop over the post list when rendering the template.

Open the `urls.py` file of your `blog` application, comment out the class-based `PostListView` URL pattern, and uncomment the `post_list` view, like this:

```
path('', views.post_list, name='post_list'),
# path('', views.PostListView.as_view(), name='post_list'),
```

Add the following additional URL pattern to list posts by tag:

```
path('tag/<slug:tag_slug>/',
     views.post_list, name='post_list_by_tag'),
```

As you can see, both patterns point to the same view, but they have different names. The first pattern will call the `post_list` view without any optional parameters, whereas the second pattern will call the view with the `tag_slug` parameter. You use a `slug` path converter to match the parameter as a lowercase string with ASCII letters or numbers, plus the hyphen and underscore characters.

The `urls.py` file of the `blog` application should now look like this:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
         views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
]
```

Since you are using the `post_list` view, edit the `blog/post/list.html` template and modify the pagination to use the `posts` object:

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

Add the following lines highlighted in bold to the `blog/post/list.html` template:

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

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

{% block content %}
  <h1>My Blog</h1>
  {% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
  {% endif %}
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

If a user is accessing the blog, they will see the list of all posts. If they filter by posts tagged with a specific tag, they will see the tag that they are filtering by.

Now, edit the `blog/post/list.html` template and change the way tags are displayed, as follows. New lines are highlighted in bold:

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

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

{% block content %}
  <h1>My Blog</h1>
  {% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
  {% endif %}
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
```

```
        </a>
      </h2>
      <p class="tags">
        Tags:
        {% for tag in post.tags.all %}
          <a href="{% url "blog:post_list_by_tag" tag.slug %}">
            {{ tag.name }}
          </a>
          {% if not forloop.last %}, {% endif %}
        {% endfor %}
      </p>
      <p class="date">
        Published {{ post.publish }} by {{ post.author }}
      </p>
      {{ post.body|truncatewords:30|linebreaks }}
    {% endfor %}
    {% include "pagination.html" with page=posts %}
{% endblock %}
```

In the preceding code, we loop through all the tags of a post displaying a custom link to the URL to filter posts by that tag. We build the URL with `{% url "blog:post_list_by_tag" tag.slug %}`, using the name of the URL and the `slug` tag as its parameter. You separate the tags by commas.

Open `http://127.0.0.1:8000/blog/tag/jazz/` in your browser. You will see the list of posts filtered by that tag, like this:

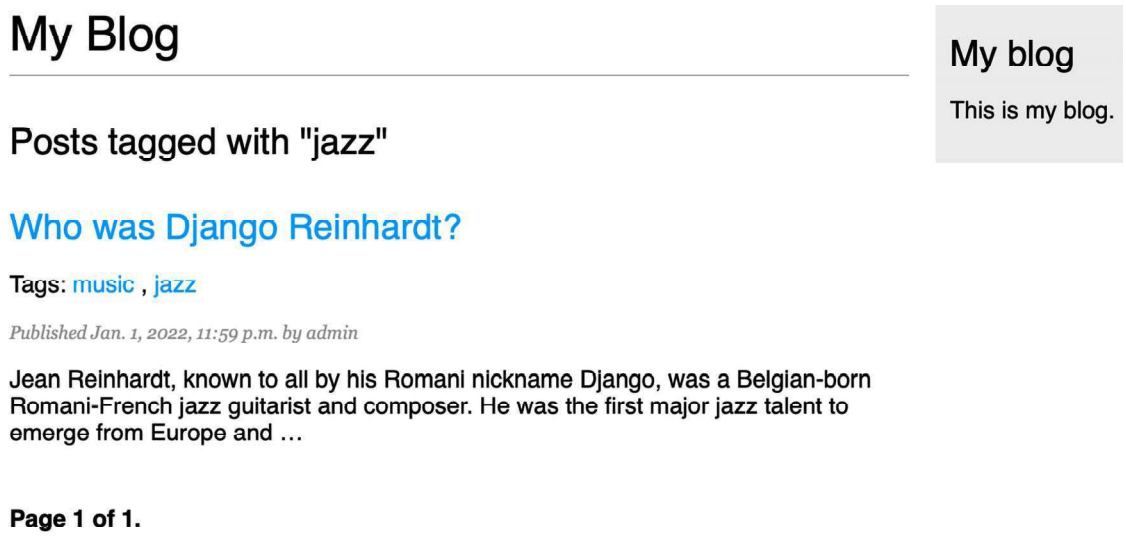

Figure 3.6: A post filtered by the tag "jazz"

Retrieving posts by similarity

Now that we have implemented tagging for blog posts, you can do many interesting things with tags. Tags allow you to categorize posts in a non-hierarchical manner. Posts about similar topics will have several tags in common. We will build a functionality to display similar posts by the number of tags they share. In this way, when a user reads a post, we can suggest to them that they read other related posts.

In order to retrieve similar posts for a specific post, you need to perform the following steps:

1. Retrieve all tags for the current post
2. Get all posts that are tagged with any of those tags
3. Exclude the current post from that list to avoid recommending the same post
4. Order the results by the number of tags shared with the current post
5. In the case of two or more posts with the same number of tags, recommend the most recent post
6. Limit the query to the number of posts you want to recommend

These steps are translated into a complex QuerySet that you will include in your `post_detail` view.

Open the `views.py` file of your `blog` application and add the following import at the top of it:

```
from django.db.models import Count
```

This is the `Count` aggregation function of the Django ORM. This function will allow you to perform aggregated counts of tags. `django.db.models` includes the following aggregation functions:

- `Avg`: The mean value
- `Max`: The maximum value
- `Min`: The minimum value
- `Count`: The total number of objects

You can learn about aggregation at `https://docs.djangoproject.com/en/4.1/topics/db/aggregation/`.

Open the `views.py` file of your `blog` application and add the following lines to the `post_detail` view. New lines are highlighted in bold:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)

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

```
    # Form for users to comment
    form = CommentForm()

    # List of similar posts
    post_tags_ids = post.tags.values_list('id', flat=True)
    similar_posts = Post.published.filter(tags__in=post_tags_ids)\
                                  .exclude(id=post.id)
    similar_posts = similar_posts.annotate(same_tags=Count('tags'))\
                                 .order_by('-same_tags','-publish')[:4]

    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                   'comments': comments,
                   'form': form,
                   'similar_posts': similar_posts})
```

The preceding code is as follows:

1. You retrieve a Python list of IDs for the tags of the current post. The `values_list()` QuerySet returns tuples with the values for the given fields. You pass `flat=True` to it to get single values such as `[1, 2, 3, ...]` instead of one-tuples such as `[(1,), (2,), (3,) ...]`.
2. You get all posts that contain any of these tags, excluding the current post itself.
3. You use the `Count` aggregation function to generate a calculated field—`same_tags`—that contains the number of tags shared with all the tags queried.
4. You order the result by the number of shared tags (descending order) and by `publish` to display recent posts first for the posts with the same number of shared tags. You slice the result to retrieve only the first four posts.
5. We pass the `similar_posts` object to the context dictionary for the `render()` function.

Now, edit the `blog/post/detail.html` template and add the following code highlighted in bold:

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

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
```

```
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>

  <h2>Similar posts</h2>
  {% for post in similar_posts %}
    <p>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </p>
  {% empty %}
    There are no similar posts yet.
  {% endfor %}

  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
  {% for comment in comments %}
    <div class="comment">
      <p class="info">
        Comment {{ forloop.counter }} by {{ comment.name }}
        {{ comment.created }}
      </p>
      {{ comment.body|linebreaks }}
    </div>
  {% empty %}
    <p>There are no comments yet.</p>
  {% endfor %}
  {% include "blog/post/includes/comment_form.html" %}
{% endblock %}
```

The post detail page should look like this:

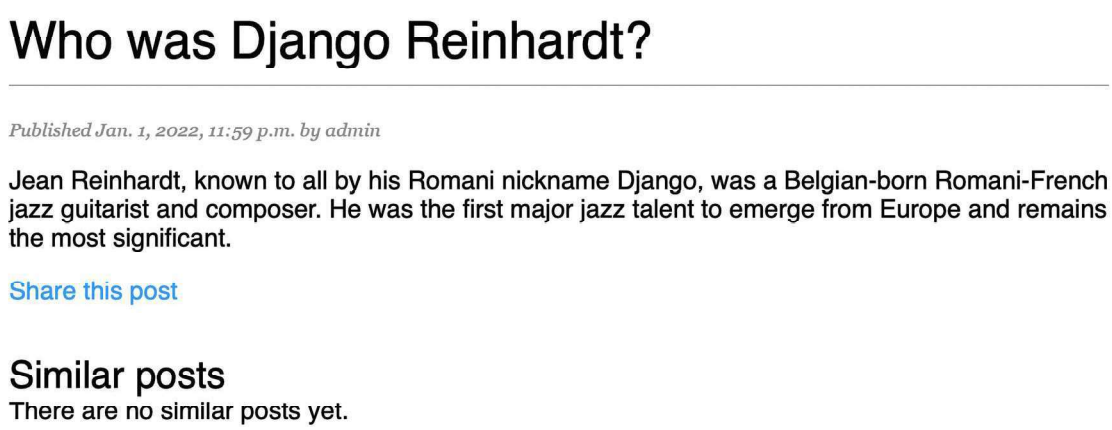

Figure 3.7: The post detail page, including a list of similar posts

Open `http://127.0.0.1:8000/admin/blog/post/` in your browser, edit a post that has no tags, and add the `music` and `jazz` tags as follows:

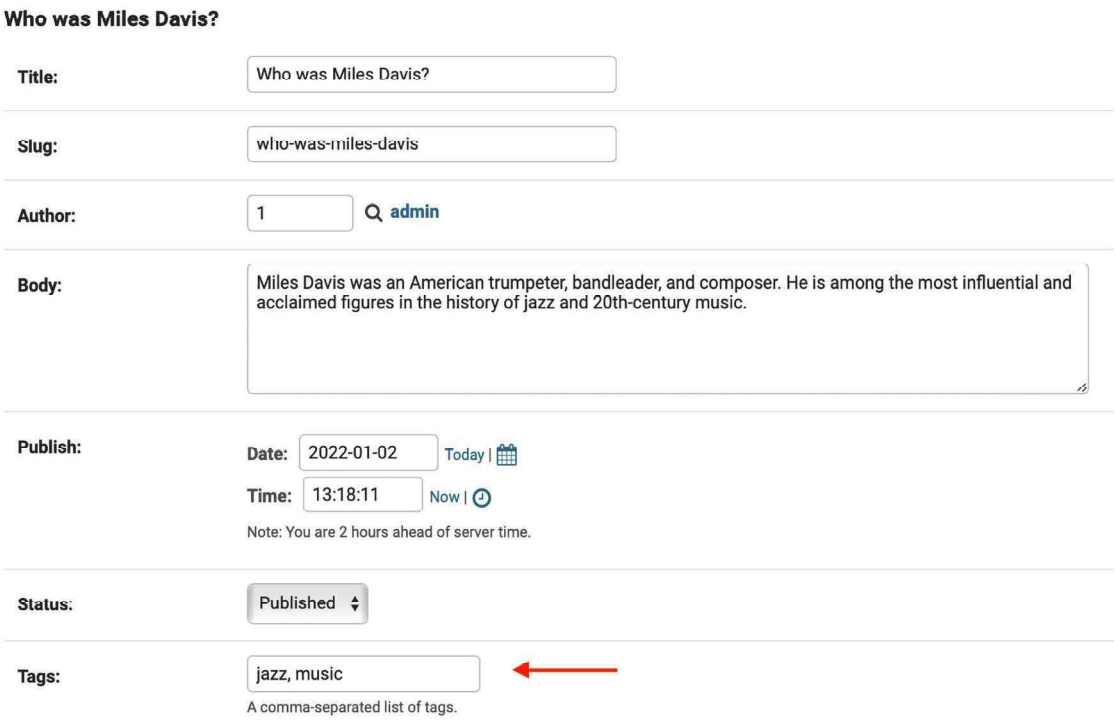

Figure 3.8: Adding the "jazz" and "music" tags to a post

Edit another post and add the `jazz` tag as follows:

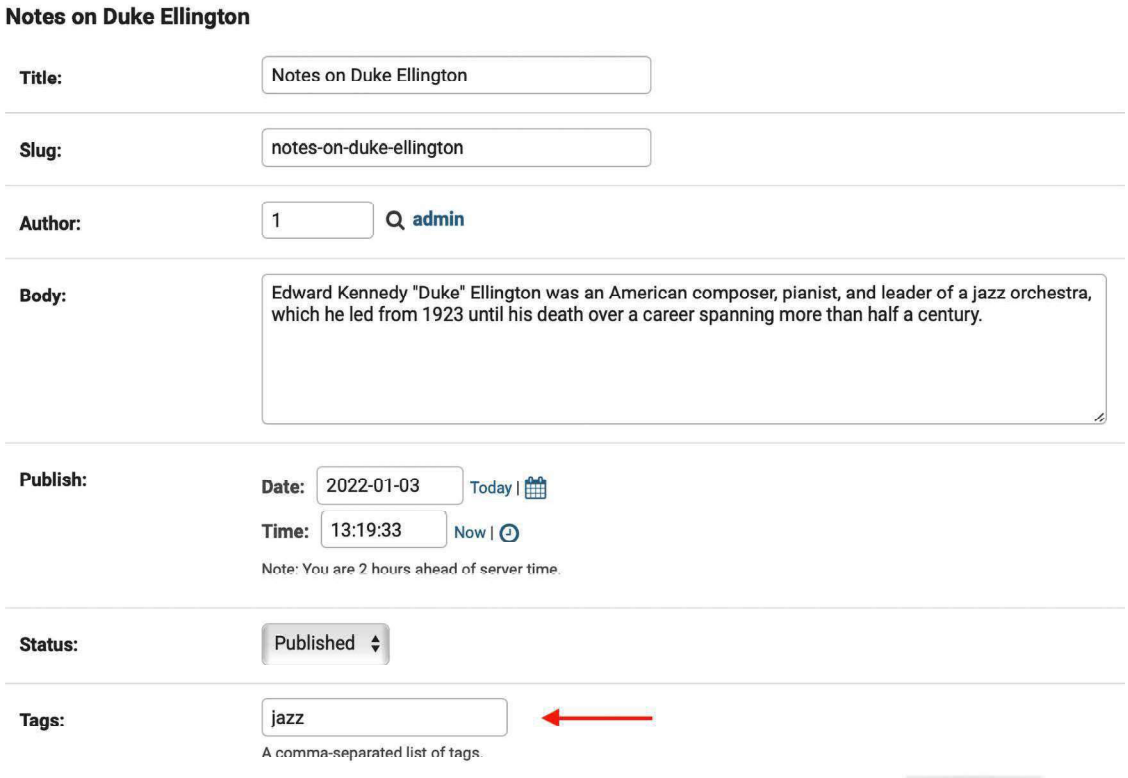

Figure 3.9: Adding the "jazz" tag to a post

The post detail page for the first post should now look like this:

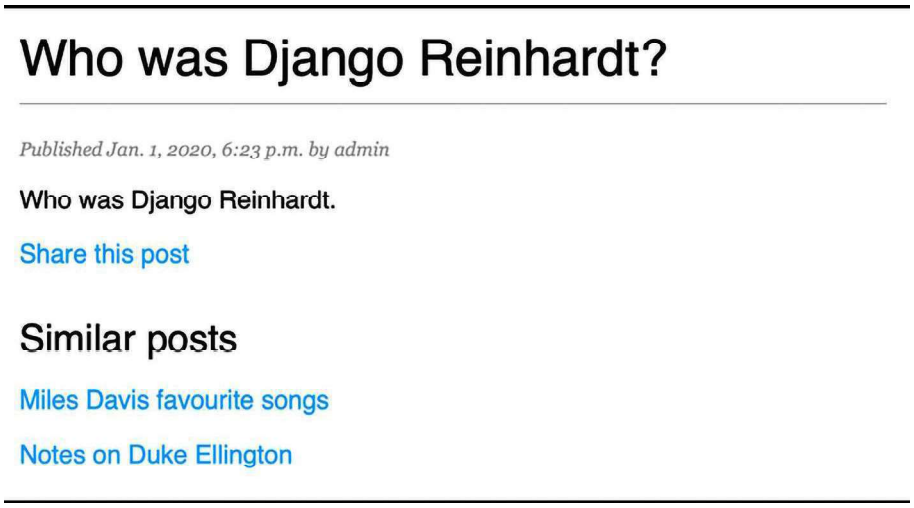

Figure 3.10: The post detail page, including a list of similar posts

The posts recommended in the **Similar posts** section of the page appear in descending order based on the number of shared tags with the original post.

We are now able to successfully recommend similar posts to the readers. `django-taggit` also includes a `similar_objects()` manager that you can use to retrieve objects by shared tags. You can take a look at all `django-taggit` managers at `https://django-taggit.readthedocs.io/en/latest/api.html`.

You can also add the list of tags to your post detail template in the same way as you did in the `blog/post/list.html` template.

Creating custom template tags and filters

Django offers a variety of built-in template tags, such as `{% if %}` or `{% block %}`. You used different template tags in *Chapter 1*, *Building a Blog Application*, and *Chapter 2*, *Enhancing Your Blog with Advanced Features*. You can find a complete reference of built-in template tags and filters at `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/`.

Django also allows you to create your own template tags to perform custom actions. Custom template tags come in very handy when you need to add a functionality to your templates that is not covered by the core set of Django template tags. This can be a tag to execute a QuerySet or any server-side processing that you want to reuse across templates. For example, we could build a template tag to display the list of latest posts published on the blog. We could include this list in the sidebar, so that it is always visible, regardless of the view that processes the request.

Implementing custom template tags

Django provides the following helper functions that allow you to easily create template tags:

- `simple_tag`: Processes the given data and returns a string
- `inclusion_tag`: Processes the given data and returns a rendered template

Template tags must live inside Django applications.

Inside your `blog` application directory, create a new directory, name it `templatetags`, and add an empty `__init__.py` file to it. Create another file in the same folder and name it `blog_tags.py`. The file structure of the blog application should look like the following:

```
blog/
    __init__.py
    models.py
    ...
    templatetags/
        __init__.py
        blog_tags.py
```

The way you name the file is important. You will use the name of this module to load tags in templates.

Creating a simple template tag

Let's start by creating a simple tag to retrieve the total posts that have been published on the blog.

Edit the `templatetags/blog_tags.py` file you just created and add the following code:

```
from django import template
from ..models import Post

register = template.Library()

@register.simple_tag
def total_posts():
    return Post.published.count()
```

We have created a simple template tag that returns the number of posts published in the blog.

Each module that contains template tags needs to define a variable called `register` to be a valid tag library. This variable is an instance of `template.Library`, and it's used to register the template tags and filters of the application.

In the preceding code, we have defined a tag called `total_posts` with a simple Python function. We have added the `@register.simple_tag` decorator to the function, to register it as a simple tag. Django will use the function's name as the tag name. If you want to register it using a different name, you can do so by specifying a `name` attribute, such as `@register.simple_tag(name='my_tag')`.

After adding a new template tags module, you will need to restart the Django development server in order to use the new tags and filters in templates.

Before using custom template tags, we have to make them available for the template using the `{% load %}` tag. As mentioned before, we need to use the name of the Python module containing your template tags and filters.

Edit the `blog/templates/base.html` template and add `{% load blog_tags %}` at the top of it to load your template tags module. Then, use the tag you created to display your total posts, as follows. The new lines are highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
```

```
    <link href="{% static "css/blog.css" %}" rel="stylesheet">
  </head>
  <body>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
    <div id="sidebar">
      <h2>My blog</h2>
      <p>
        This is my blog.
        I've written {% total_posts %} posts so far.
      </p>
    </div>
  </body>
  </html>
```

You will need to restart the server to keep track of the new files added to the project. Stop the development server with *Ctrl* + *C* and run it again using the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/blog/` in your browser. You should see the total number of posts in the sidebar of the site, as follows:

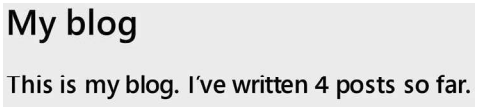

Figure 3.11: The total posts published included in the sidebar

If you see the following error message, it's very likely you didn't restart the development server:

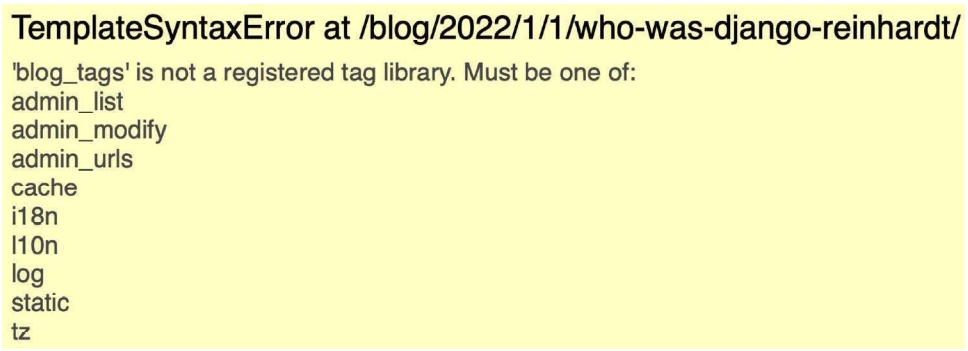

Figure 3.12: The error message when a template tag library is not registered

Template tags allow you to process any data and add it to any template regardless of the view executed. You can perform QuerySets or process any data to display results in your templates.

Creating an inclusion template tag

We will create another tag to display the latest posts in the sidebar of the blog. This time, we will implement an inclusion tag. Using an inclusion tag, you can render a template with context variables returned by your template tag.

Edit the `templatetags/blog_tags.py` file and add the following code:

```
@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[:count]
    return {'latest_posts': latest_posts}
```

In the preceding code, we have registered the template tag using the `@register.inclusion_tag` decorator. We have specified the template that will be rendered with the returned values using `blog/post/latest_posts.html`. The template tag will accept an optional `count` parameter that defaults to `5`. This parameter will allow us to specify the number of posts to display. We use this variable to limit the results of the query `Post.published.order_by('-publish')[:count]`.

Note that the function returns a dictionary of variables instead of a simple value. Inclusion tags have to return a dictionary of values, which is used as the context to render the specified template. The template tag we just created allows us to specify the optional number of posts to display as `{% show_latest_posts 3 %}`.

Now, create a new template file under `blog/post/` and name it `latest_posts.html`.

Edit the new `blog/post/latest_posts.html` template and add the following code to it:

```
<ul>
  {% for post in latest_posts %}
    <li>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </li>
  {% endfor %}
</ul>
```

In the preceding code, you display an unordered list of posts using the `latest_posts` variable returned by your template tag. Now, edit the `blog/base.html` template and add the new template tag to display the last three posts, as follows. The new lines are highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
```

```
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>
      This is my blog.
      I've written {% total_posts %} posts so far.
    </p>
    <h3>Latest posts</h3>
    {% show_latest_posts 3 %}
  </div>
</body>
</html>
```

The template tag is called, passing the number of posts to display, and the template is rendered in place with the given context.

Next, return to your browser and refresh the page. The sidebar should now look like this:

Figure 3.13: The blog sidebar, including the latest published posts

Creating a template tag that returns a QuerySet

Finally, we will create a simple template tag that returns a value. We will store the result in a variable that can be reused, rather than outputting it directly. We will create a tag to display the most commented posts.

Edit the `templatetags/blog_tags.py` file and add the following import and template tag to it:

```
from django.db.models import Count

@register.simple_tag
def get_most_commented_posts(count=5):
    return Post.published.annotate(
                total_comments=Count('comments')
            ).order_by('-total_comments')[:count]
```

In the preceding template tag, you build a QuerySet using the `annotate()` function to aggregate the total number of comments for each post. You use the `Count` aggregation function to store the number of comments in the computed `total_comments` field for each `Post` object. You order the QuerySet by the computed field in descending order. You also provide an optional `count` variable to limit the total number of objects returned.

In addition to `Count`, Django offers the aggregation functions `Avg`, `Max`, `Min`, and `Sum`. You can read more about aggregation functions at `https://docs.djangoproject.com/en/4.1/topics/db/aggregation/`.

Next, edit the `blog/base.html` template and add the following code highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>
      This is my blog.
      I've written {% total_posts %} posts so far.
    </p>
    <h3>Latest posts</h3>
    {% show_latest_posts 3 %}
    <h3>Most commented posts</h3>
```

```
    {% get_most_commented_posts as most_commented_posts %}
    <ul>
      {% for post in most_commented_posts %}
        <li>
          <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
</body>
</html>
```

In the preceding code, we store the result in a custom variable using the `as` argument followed by the variable name. For the template tag, we use `{% get_most_commented_posts as most_commented_posts %}` to store the result of the template tag in a new variable named `most_commented_posts`. Then, we display the returned posts using an HTML unordered list element.

Now open your browser and refresh the page to see the final result. It should look like the following:

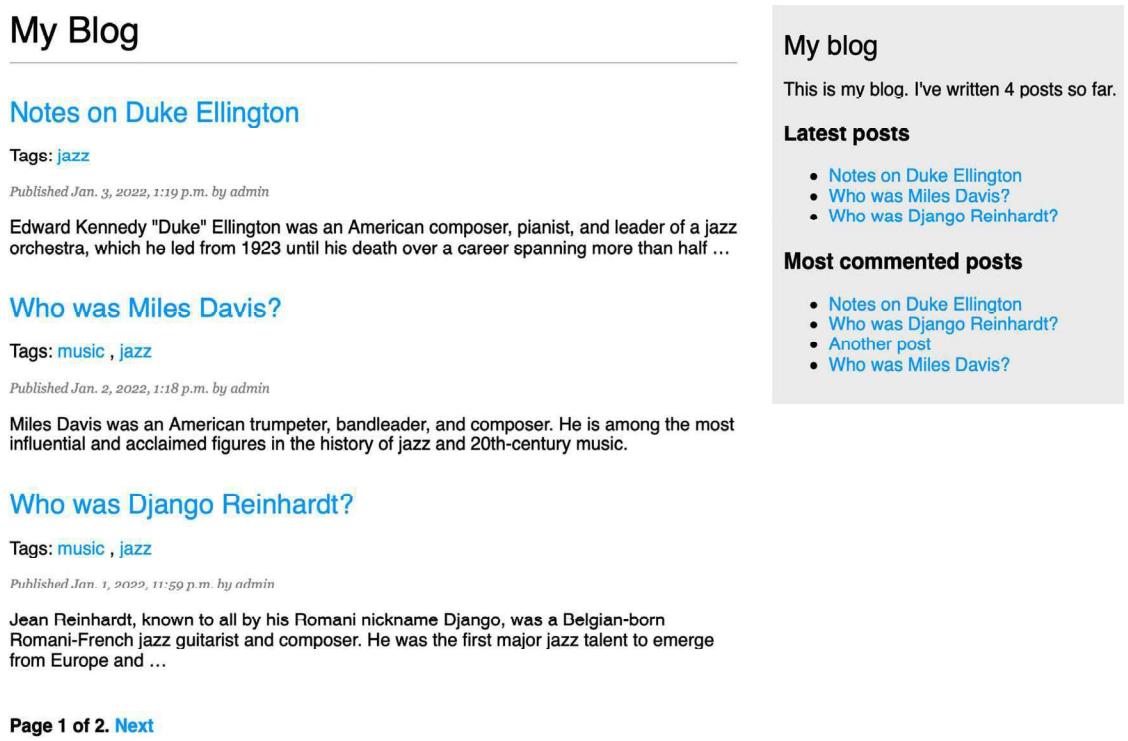

Figure 3.14: The post list view, including the complete sidebar with the latest and most commented posts

You have now a clear idea about how to build custom template tags. You can read more about them at `https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/`.

Implementing custom template filters

Django has a variety of built-in template filters that allow you to alter variables in templates. These are Python functions that take one or two parameters, the value of the variable that the filter is applied to, and an optional argument. They return a value that can be displayed or treated by another filter.

A filter is written like `{{ variable|my_filter }}`. Filters with an argument are written like `{{ variable|my_filter:"foo" }}`. For example, you can use the `capfirst` filter to capitalize the first character of the value, like `{{ value|capfirst }}`. If `value` is `django`, the output will be `Django`. You can apply as many filters as you like to a variable, for example, `{{ variable|filter1|filter2 }}`, and each filter will be applied to the output generated by the preceding filter.

You can find the list of Django's built-in template filters at `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#built-in-filter-reference`.

Creating a template filter to support Markdown syntax

We will create a custom filter to enable you to use Markdown syntax in your blog posts and then convert the post body to HTML in the templates.

Markdown is a plain text formatting syntax that is very simple to use, and it's intended to be converted into HTML. You can write posts using simple Markdown syntax and get the content automatically converted into HTML code. Learning Markdown syntax is much easier than learning HTML. By using Markdown, you can get other non-tech savvy contributors to easily write posts for your blog. You can learn the basics of the Markdown format at `https://daringfireball.net/projects/markdown/basics`.

First, install the Python `markdown` module via `pip` using the following command in the shell prompt:

```
pip install markdown==3.4.1
```

Then, edit the `templatetags/blog_tags.py` file and include the following code:

```
from django.utils.safestring import mark_safe
import markdown

@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

We register template filters in the same way as template tags. To prevent a name clash between the function name and the `markdown` module, we have named the function `markdown_format` and we have named the filter `markdown` for use in templates, such as `{{ variable|markdown }}`.

Django escapes the HTML code generated by filters; characters of HTML entities are replaced with their HTML encoded characters. For example, `<p>` is converted to `&lt;p&gt;` (*less than* symbol, *p* character, *greater than* symbol).

We use the `mark_safe` function provided by Django to mark the result as safe HTML to be rendered in the template. By default, Django will not trust any HTML code and will escape it before placing it in the output. The only exceptions are variables that are marked as safe from escaping. This behavior prevents Django from outputting potentially dangerous HTML and allows you to create exceptions for returning safe HTML.

Edit the `blog/post/detail.html` template and add the following new code highlighted in bold:

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

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|markdown }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>

  <h2>Similar posts</h2>
  {% for post in similar_posts %}
    <p>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </p>
  {% empty %}
    There are no similar posts yet.
  {% endfor %}

  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
  {% for comment in comments %}
    <div class="comment">
      <p class="info">
```

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

  {% include "blog/post/includes/comment_form.html" %}
{% endblock %}
```

We have replaced the `linebreaks` filter of the `{{ post.body }}` template variable with the `markdown` filter. This filter will not only transform line breaks into `<p>` tags; it will also transform Markdown formatting into HTML.

Edit the `blog/post/list.html` template and add the following new code highlighted in bold:

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

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

{% block content %}
  <h1>My Blog</h1>
  {% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
  {% endif %}
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="tags">
      Tags:
      {% for tag in post.tags.all %}
        <a href="{% url "blog:post_list_by_tag" tag.slug %}">
          {{ tag.name }}
        </a>
        {% if not forloop.last %}, {% endif %}
      {% endfor %}
```

```
    </p>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|markdown|truncatewords_html:30 }}
  {% endfor %}
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

We have added the new `markdown` filter to the `{{ post.body }}` template variable. This filter will transform the Markdown content into HTML. Therefore, we have replaced the previous `truncatewords` filter with the `truncatewords_html` filter. This filter truncates a string after a certain number of words avoiding unclosed HTML tags.

Now open `http://127.0.0.1:8000/admin/blog/post/add/` in your browser and create a new post with the following body:

```
This is a post formatted with markdown
--------------------------------------

*This is emphasized* and **this is more emphasized**.

Here is a list:

* One
* Two
* Three

And a [link to the Django website](https://www.djangoproject.com/).
```

The form should look like this:

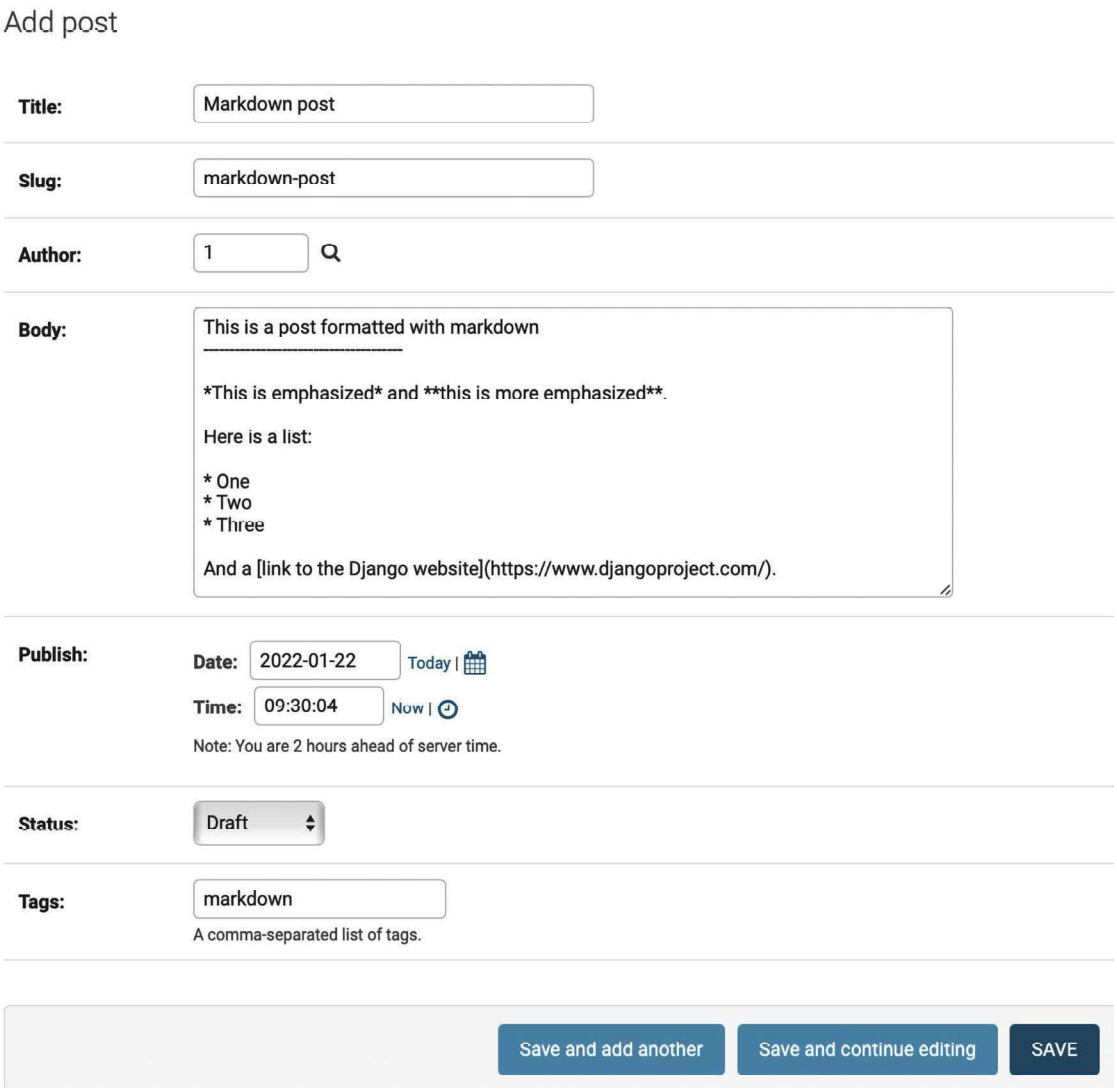

Figure 3.15: The post with Markdown content rendered as HTML

Open `http://127.0.0.1:8000/blog/` in your browser and take a look at how the new post is rendered. You should see the following output:

Figure 3.16: The post with Markdown content rendered as HTML

As you can see in *Figure 3.16*, custom template filters are very useful for customizing formatting. You can find more information about custom filters at `https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/#writing-custom-template-filters`.

Adding a sitemap to the site

Django comes with a sitemap framework, which allows you to generate sitemaps for your site dynamically. A sitemap is an XML file that tells search engines the pages of your website, their relevance, and how frequently they are updated. Using a sitemap will make your site more visible in search engine rankings because it helps crawlers to index your website's content.

The Django sitemap framework depends on `django.contrib.sites`, which allows you to associate objects to particular websites that are running with your project. This comes in handy when you want to run multiple sites using a single Django project. To install the sitemap framework, we will need to activate both the `sites` and the `sitemap` applications in your project.

Edit the `settings.py` file of the project and add `django.contrib.sites` and `django.contrib.sitemaps` to the `INSTALLED_APPS` setting. Also, define a new setting for the site ID, as follows. New code is highlighted in bold:

```
# ...

SITE_ID = 1

# Application definition

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

Now, run the following command from the shell prompt to create the tables of the Django site application in the database:

```
python manage.py migrate
```

You should see an output that contains the following lines:

```
Applying sites.0001_initial... OK
Applying sites.0002_alter_domain_unique... OK
```

The `sites` application is now synced with the database.

Next, create a new file inside your `blog` application directory and name it `sitemaps.py`. Open the file and add the following code to it:

```
from django.contrib.sitemaps import Sitemap
from .models import Post

class PostSitemap(Sitemap):
    changefreq = 'weekly'
    priority = 0.9
```

```
    def items(self):
        return Post.published.all()

    def lastmod(self, obj):
        return obj.updated
```

We have defined a custom sitemap by inheriting the `Sitemap` class of the `sitemaps` module. The `changefreq` and `priority` attributes indicate the change frequency of your post pages and their relevance in your website (the maximum value is `1`).

The `items()` method returns the QuerySet of objects to include in this sitemap. By default, Django calls the `get_absolute_url()` method on each object to retrieve its URL. Remember that we implemented this method in *Chapter 2, Enhancing Your Blog with Advanced Features*, to define the canonical URL for posts. If you want to specify the URL for each object, you can add a `location` method to your sitemap class.

The `lastmod` method receives each object returned by `items()` and returns the last time the object was modified.

Both the `changefreq` and `priority` attributes can be either methods or attributes. You can take a look at the complete sitemap reference in the official Django documentation located at `https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/`.

We have created the sitemap. Now we just need to create an URL for it.

Edit the main `urls.py` file of the `mysite` project and add the sitemap, as follows. New lines are highlighted in bold:

```
from django.urls import path, include
from django.contrib import admin
from django.contrib.sitemaps.views import sitemap
from blog.sitemaps import PostSitemap

sitemaps = {
    'posts': PostSitemap,
}

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
    path('sitemap.xml', sitemap, {'sitemaps': sitemaps},
         name='django.contrib.sitemaps.views.sitemap')
]
```

In the preceding code, we have included the required imports and have defined a `sitemaps` dictionary. Multiple sitemaps can be defined for the site. We have defined a URL pattern that matches with the `sitemap.xml` pattern and uses the `sitemap` view provided by Django. The `sitemaps` dictionary is passed to the `sitemap` view.

Start the development from the shell prompt with the following command:

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/sitemap.xml` in your browser. You will see an XML output including all of the published posts like this:

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>http://example.com/blog/2022/1/22/markdown-post/</loc>
    <lastmod>2022-01-22</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/3/notes-on-duke-ellington/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreqa>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/2/who-was-miles-davis/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/who-was-django-reinhardt/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/another-post/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

The URL for each `Post` object is built by calling its `get_absolute_url()` method.

The `lastmod` attribute corresponds to the post `updated` date field, as you specified in your sitemap, and the `changefreq` and `priority` attributes are also taken from the `PostSitemap` class.

The domain used to build the URLs is `example.com`. This domain comes from a `Site` object stored in the database. This default object was created when you synced the site's framework with your database. You can read more about the `sites` framework at `https://docs.djangoproject.com/en/4.1/ref/contrib/sites/`.

Open `http://127.0.0.1:8000/admin/sites/site/` in your browser. You should see something like this:

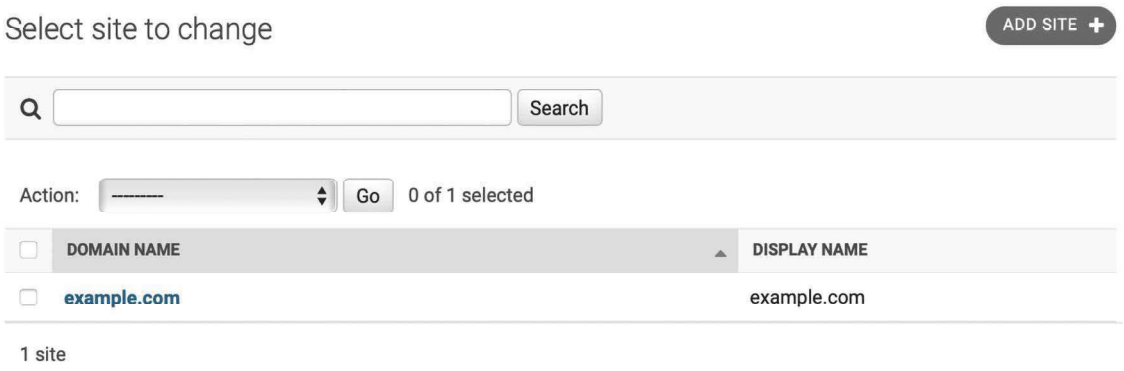

Figure 3.17: The Django administration list view for the Site model of the site's framework

Figure 3.17 contains the list display administration view for the site's framework. Here, you can set the domain or host to be used by the site's framework and the applications that depend on it. To generate URLs that exist in your local environment, change the domain name to `localhost:8000`, as shown in *Figure 3.18*, and save it:

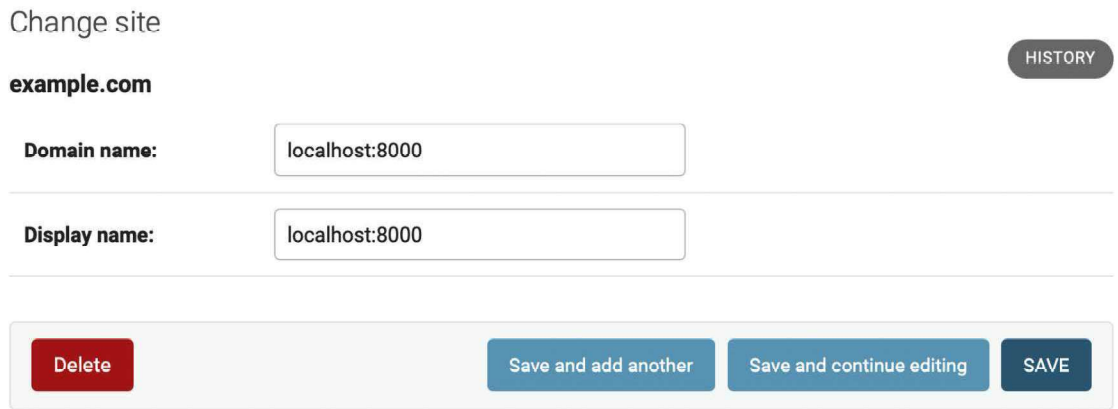

Figure 3.18: The Django administration edit view for the Site model of the site's framework

Open `http://127.0.0.1:8000/sitemap.xml` in your browser again. The URLs displayed in your feed will now use the new hostname and look like `http://localhost:8000/blog/2022/1/22/markdown-post/`. Links are now accessible in your local environment. In a production environment, you will have to use your website's domain to generate absolute URLs.

Creating feeds for blog posts

Django has a built-in syndication feed framework that you can use to dynamically generate RSS or Atom feeds in a similar manner to creating sitemaps using the site's framework. A web feed is a data format (usually XML) that provides users with the most recently updated content. Users can subscribe to the feed using a feed aggregator, a software that is used to read feeds and get new content notifications.

Create a new file in your `blog` application directory and name it `feeds.py`. Add the following lines to it:

```
import markdown
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords_html
from django.urls import reverse_lazy
from .models import Post

class LatestPostsFeed(Feed):
    title = 'My blog'
    link = reverse_lazy('blog:post_list')
    description = 'New posts of my blog.'

    def items(self):
        return Post.published.all()[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return truncatewords_html(markdown.markdown(item.body), 30)

    def item_pubdate(self, item):
        return item.publish
```

In the preceding code, we have defined a feed by subclassing the `Feed` class of the syndication framework. The `title`, `link`, and `description` attributes correspond to the `<title>`, `<link>`, and `<description>` RSS elements, respectively.

We use `reverse_lazy()` to generate the URL for the `link` attribute. The `reverse()` method allows you to build URLs by their name and pass optional parameters. We used `reverse()` in *Chapter 2, Enhancing Your Blog with Advanced Features*.

The `reverse_lazy()` utility function is a lazily evaluated version of `reverse()`. It allows you to use a URL reversal before the project's URL configuration is loaded.

The `items()` method retrieves the objects to be included in the feed. We retrieve the last five published posts to include them in the feed.

The `item_title()`, `item_description()`, and `item_pubdate()` methods will receive each object returned by `items()` and return the title, description and publication date for each item.

In the `item_description()` method, we use the `markdown()` function to convert Markdown content to HTML and the `truncatewords_html()` template filter function to cut the description of posts after 30 words, avoiding unclosed HTML tags.

Now, edit the `blog/urls.py` file, import the `LatestPostsFeed` class, and instantiate the feed in a new URL pattern, as follows. New lines are highlighted in bold:

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

app_name = 'blog'

urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
         views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
    path('feed/', LatestPostsFeed(), name='post_feed'),
]
```

Navigate to `http://127.0.0.1:8000/blog/feed/` in your browser. You should now see the RSS feed, including the last five blog posts:

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
  <channel>
    <title>My blog</title>
```

```
    <link>http://localhost:8000/blog/</link>
    <description>New posts of my blog.</description>
    <atom:link href="http://localhost:8000/blog/feed/" rel="self"/>
    <language>en-us</language>
    <lastBuildDate>Fri, 2 Jan 2020 09:56:40 +0000</lastBuildDate>
    <item>
      <title>Who was Django Reinhardt?</title>
      <link>http://localhost:8000/blog/2020/1/2/who-was-django-
      reinhardt/</link>
      <description>Who was Django Reinhardt.</description>
      <guid>http://localhost:8000/blog/2020/1/2/who-was-django-
      reinhardt/</guid>
    </item>
    ...
  </channel>
</rss>
```

If you use Chrome, you will see the XML code. If you use Safari, it will ask you to install an RSS feed reader.

Let's install an RSS desktop client to view the RSS feed with a user-friendly interface. We will use Fluent Reader, which is a multi-platform RSS reader.

Download Fluent Reader for Linux, macOS, or Windows from `https://github.com/yang991178/fluent-reader/releases`.

Install Fluent Reader and open it. You will see the following screen:

Figure 3.19: Fluent Reader with no RSS feed sources

Click on the settings icon on the top right of the window. You will see a screen to add RSS feed sources like the following one:

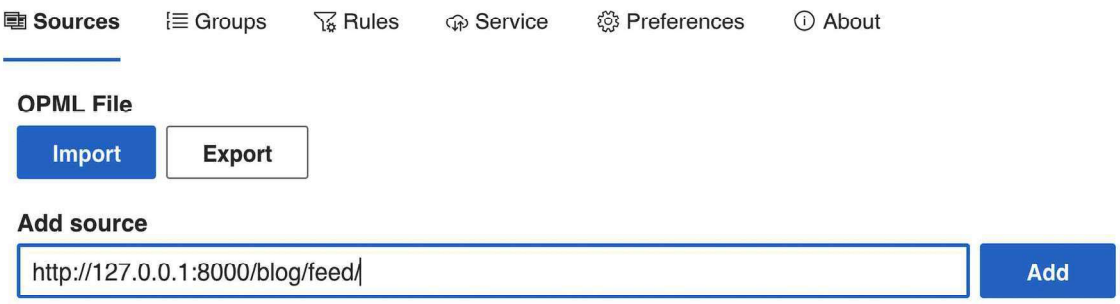

Figure 3.20: Adding an RSS feed in Fluent Reader

Enter `http://127.0.0.1:8000/blog/feed/` in the **Add source** field and click on the **Add** button.

You will see a new entry with the RSS feed of the blog in the table below the form, like this:

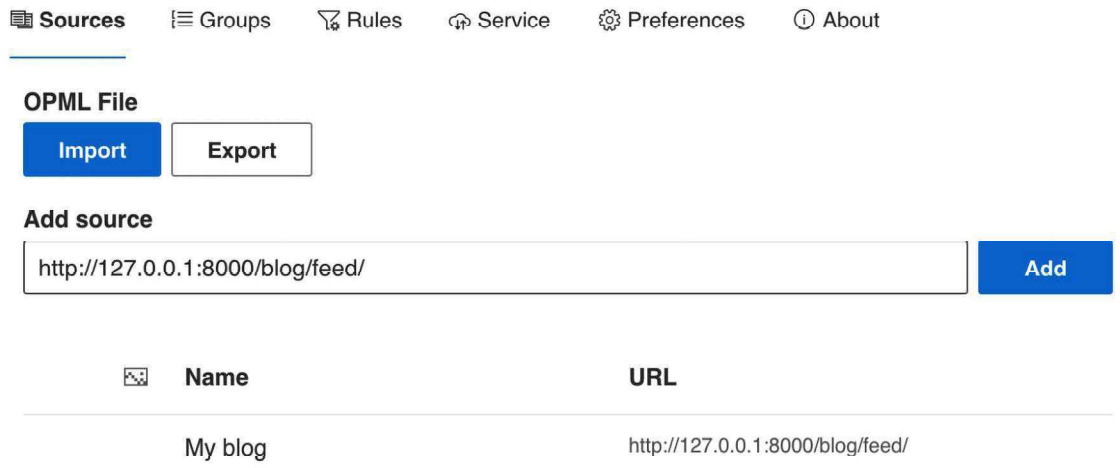

Figure 3.21: RSS feed sources in Fluent Reader

Now, go back to the main screen of Fluent Reader. You should be able to see the posts included in the blog RSS feed, as follows:

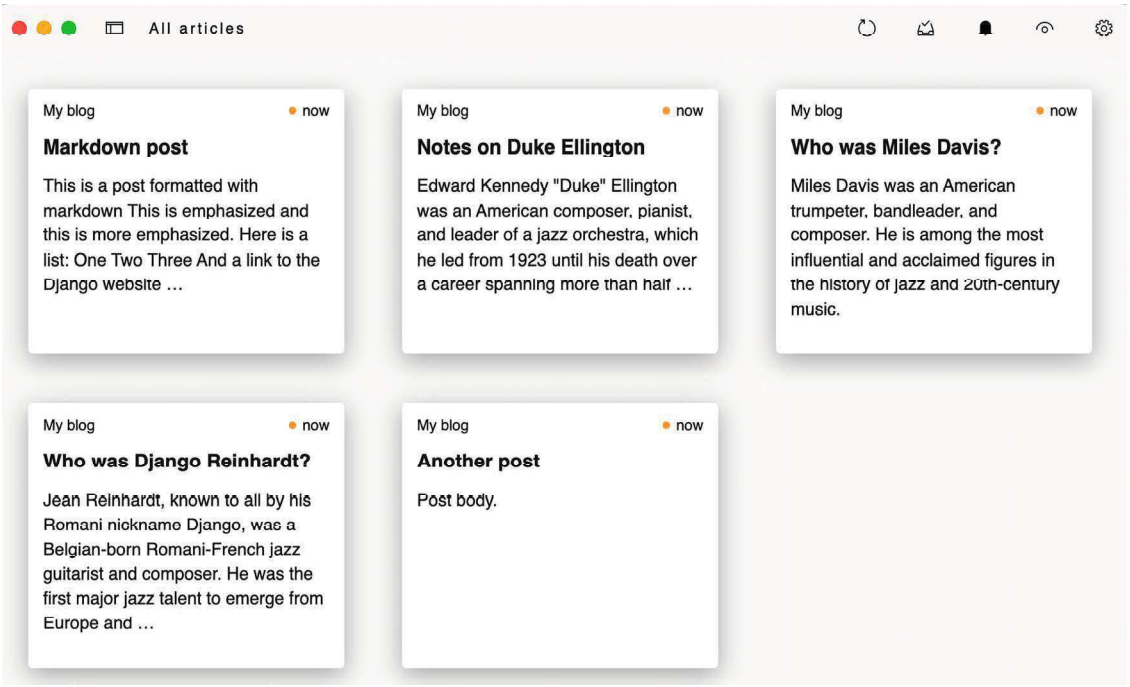

Figure 3.22: RSS feed of the blog in Fluent Reader

Click on a post to see a description:

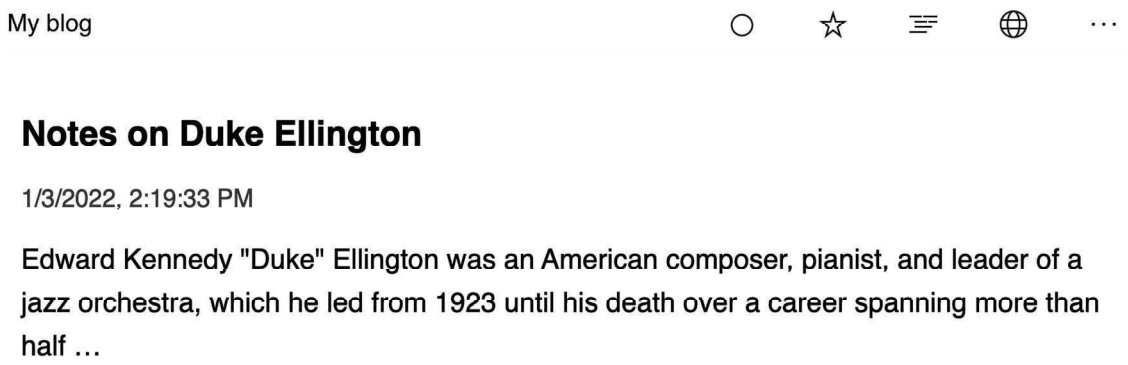

Figure 3.23: The post description in Fluent Reader

Click on the third icon at the top right of the window to load the full content of the post page:

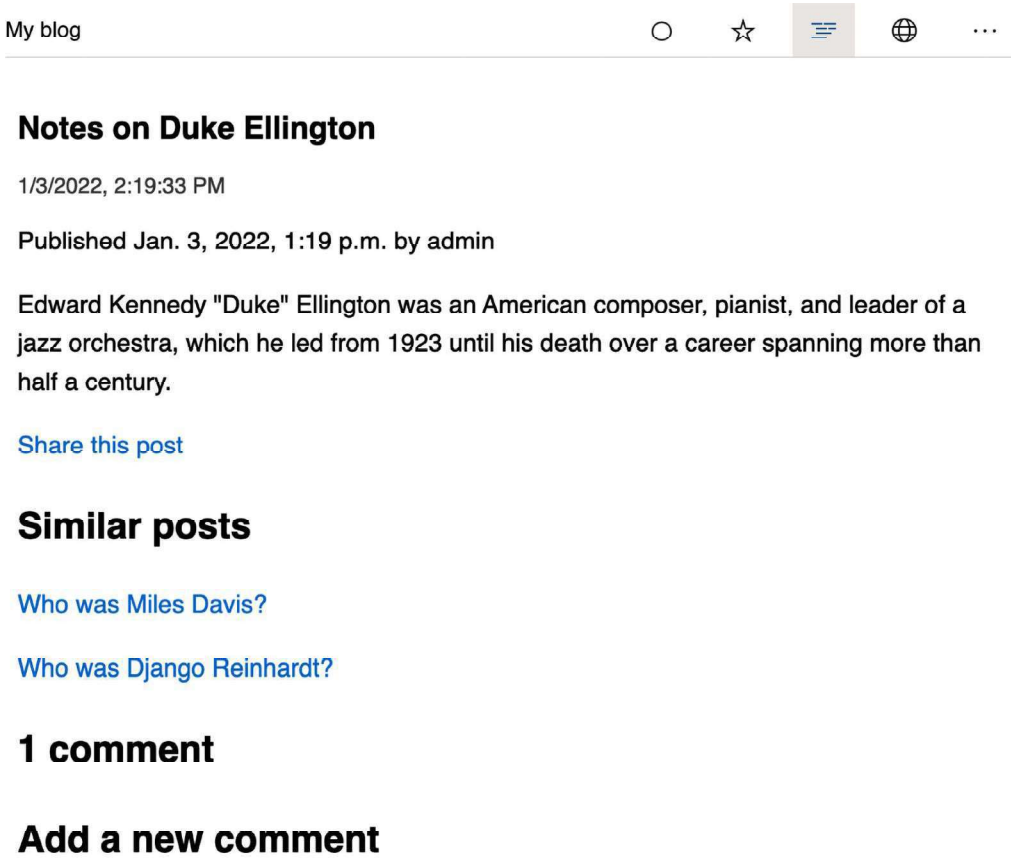

Figure 3.24: The full content of a post in Fluent Reader

The final step is to add an RSS feed subscription link to the blog's sidebar.

Open the `blog/base.html` template and add the following code highlighted in bold:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
```

```
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>
      This is my blog.
      I've written {% total_posts %} posts so far.
    </p>
    <p>
      <a href="{% url "blog:post_feed" %}">
        Subscribe to my RSS feed
      </a>
    </p>
    <h3>Latest posts</h3>
    {% show_latest_posts 3 %}
    <h3>Most commented posts</h3>
    {% get_most_commented_posts as most_commented_posts %}
    <ul>
      {% for post in most_commented_posts %}
        <li>
          <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
</body>
</html>
```

Now open `http://127.0.0.1:8000/blog/` in your browser and take a look at the sidebar. The new link will take users to the blog's feed:

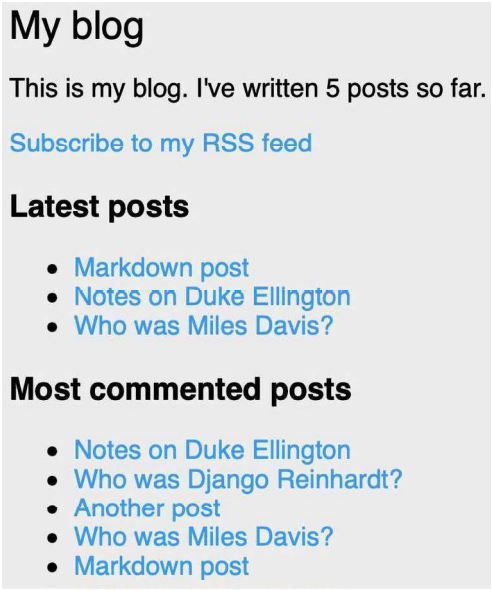

Figure 3.25: The RSS feed subscription link added to the sidebar

You can read more about the Django syndication feed framework at `https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/`.

Adding full-text search to the blog

Next, we will add search capabilities to the blog. Searching for data in the database with user input is a common task for web applications. The Django ORM allows you to perform simple matching operations using, for example, the `contains` filter (or its case-insensitive version, `icontains`). You can use the following query to find posts that contain the word `framework` in their body:

```
from blog.models import Post
Post.objects.filter(body__contains='framework')
```

However, if you want to perform complex search lookups, retrieving results by similarity, or by weighting terms based on how frequently they appear in the text or by how important different fields are (for example, relevancy of the term appearing in the title versus in the body), you will need to use a full-text search engine. When you consider large blocks of text, building queries with operations on a string of characters is not enough. A full-text search examines the actual words against stored content as it tries to match search criteria.

Django provides a powerful search functionality built on top of PostgreSQL's full-text search features. The `django.contrib.postgres` module provides functionalities offered by PostgreSQL that are not shared by the other databases that Django supports. You can learn about PostgreSQL's full-text search support at `https://www.postgresql.org/docs/14/textsearch.html`.

Although Django is a database-agnostic web framework, it provides a module that supports part of the rich feature set offered by PostgreSQL, which is not offered by other databases that Django supports.

Installing PostgreSQL

We are currently using an SQLite database for the `mysite` project. SQLite support for full-text search is limited and Django doesn't support it out of the box. However, PostgreSQL is much better suited for full-text search and we can use the `django.contrib.postgres` module to use PostgreSQL's full-text search capabilities. We will migrate our data from SQLite to PostgreSQL to benefit from its full-text search features.

SQLite is sufficient for development purposes. However, for a production environment, you will need a more powerful database, such as PostgreSQL, MariaDB, MySQL, or Oracle.

Download the PostgreSQL installer for macOS or Windows at `https://www.postgresql.org/download/`. On the same page, you can find instructions to install PostgreSQL on different Linux distributions. Follow the instructions on the website to install and run PostgreSQL.

If you are using macOS and you choose to install PostgreSQL using `Postgres.app`, you will need to configure the `$PATH` variable to use the command line tools, as explained in `https://postgresapp.com/documentation/cli-tools.html`.

You also need to install the `psycopg2` PostgreSQL adapter for Python. Run the following command in the shell prompt to install it:

```
pip install psycopg2-binary==2.9.3
```

Creating a PostgreSQL database

Let's create a user for the PostgreSQL database. We will use `psql`, which is a terminal-based frontend to PostgreSQL. Enter the PostgreSQL terminal by running the following command in the shell prompt:

```
psql
```

You will see the following output:

```
psql (14.2)
Type "help" for help.
```

Enter the following command to create a user that can create databases:

```
CREATE USER blog WITH PASSWORD 'xxxxxx';
```

Replace xxxxxx with your desired password and execute the command. You will see the following output:

```
CREATE ROLE
```

The user has been created. Let's now create a `blog` database and give ownership to the `blog` user you just created.

Execute the following command:

```
CREATE DATABASE blog OWNER blog ENCODING 'UTF8';
```

With this command we tell PostgreSQL to create a database named `blog`, we give the ownership of the database to the `blog` user we created before, and we indicate that the `UTF8` encoding has to be used for the new database. You will see the following output:

```
CREATE DATABASE
```

We have successfully created the PostgreSQL user and database.

Dumping the existing data

Before switching the database in the Django project, we need to dump the existing data from the SQLite database. We will export the data, switch the project's database to PostgreSQL, and import the data into the new database.

Django comes with a simple way to load and dump data from the database into files that are called **fixtures**. Django supports fixtures in JSON, XML, or YAML formats. We are going to create a fixture with all data contained in the database.

The `dumpdata` command dumps data from the database into the standard output, serialized in JSON format by default. The resulting data structure includes information about the model and its fields for Django to be able to load it into the database.

You can limit the output to the models of an application by providing the application names to the command, or specifying single models for outputting data using the `app.Model` format. You can also specify the format using the `--format` flag. By default, `dumpdata` outputs the serialized data to the standard output. However, you can indicate an output file using the `--output` flag. The `--indent` flag allows you to specify indentation. For more information on `dumpdata` parameters, run `python manage.py dumpdata --help`.

Execute the following command from the shell prompt:

```
python manage.py dumpdata --indent=2 --output=mysite_data.json
```

You will see an output similar to the following:

```
[..............................................]
```

All existing data has been exported in JSON format to a new file named `mysite_data.json`. You can view the file contents to see the JSON structure that includes all the different data objects for the different models of your installed applications. If you get an encoding error when running the command, include the `-Xutf8` flag as follows to activate Python UTF-8 mode:

```
python -Xutf8 manage.py dumpdata --indent=2 --output=mysite_data.json
```

We will now switch the database in the Django project and then we will import the data into the new database.

Switching the database in the project

Edit the `settings.py` file of your project and modify the `DATABASES` setting to make it look as follows. New code is highlighted in bold:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'blog',
        'USER': 'blog',
        'PASSWORD': 'xxxxxx',
    }
}
```

Replace `xxxxxx` with the password you used when creating the PostgreSQL user. The new database is empty.

Run the following command to apply all database migrations to the new PostgreSQL database:

```
python manage.py migrate
```

You will see an output, including all the migrations that have been applied, like this:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions, sites,
taggit
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 auth.0012_alter_user_first_name_max_length... OK
  Applying taggit.0001_initial... OK
  Applying taggit.0002_auto_20150616_2121... OK
  Applying taggit.0003_taggeditem_add_unique_index... OK
  Applying blog.0001_initial... OK
  Applying blog.0002_alter_post_slug... OK
  Applying blog.0003_comment... OK
  Applying blog.0004_post_tags... OK
  Applying sessions.0001_initial... OK
  Applying sites.0001_initial... OK
  Applying sites.0002_alter_domain_unique... OK
  Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
  Applying taggit.0005_auto_20220424_2025... OK
```

Loading the data into the new database

Run the following command to load the data into the PostgreSQL database:

```
python manage.py loaddata mysite_data.json
```

You will see the following output:

```
Installed 104 object(s) from 1 fixture(s)
```

The number of objects might differ, depending on the users, posts, comments, and other objects that have been created in the database.

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

```
python manage.py runserver
```

Open `http://127.0.0.1:8000/admin/blog/post/` in your browser to verify that all posts have been loaded into the new database. You should see all the posts, as follows:

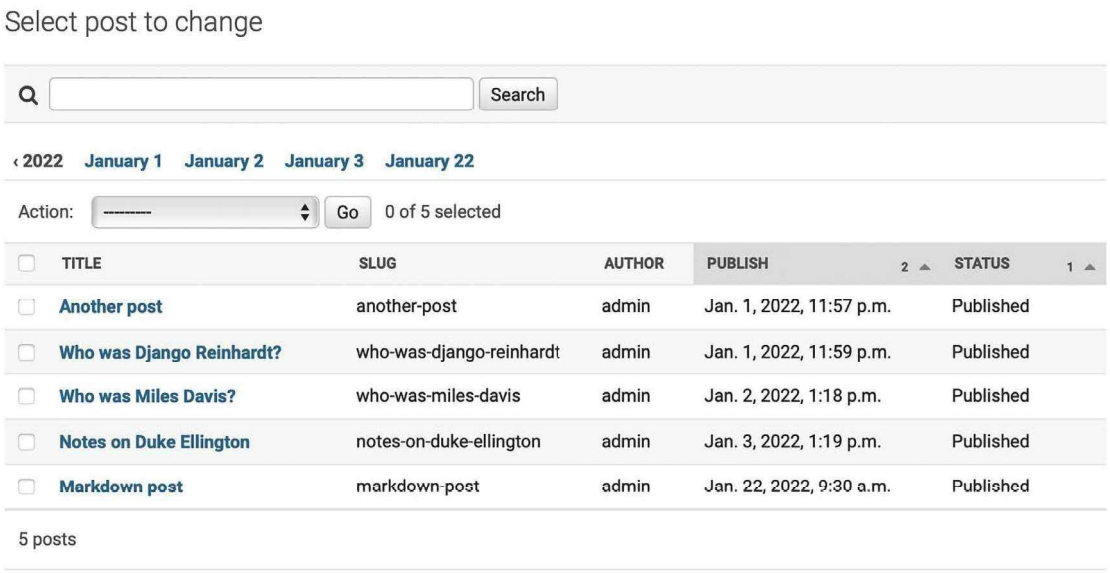

Figure 3.26: The list of posts on the administration site

Simple search lookups

Edit the `settings.py` file of your project and add `django.contrib.postgres` to the `INSTALLED_APPS` setting, as follows:

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

Open the Django shell by running the following command in the system shell prompt:

```
python manage.py shell
```

Now you can search against a single field using the `search` QuerySet lookup.

Run the following code in the Python shell:

```
>>> from blog.models import Post
>>> Post.objects.filter(title__search='django')
<QuerySet [<Post: Who was Django Reinhardt?>]>
```

This query uses PostgreSQL to create a search vector for the `body` field and a search query from the term `django`. Results are obtained by matching the query with the vector.

Searching against multiple fields

You might want to search against multiple fields. In this case, you will need to define a `SearchVector` object. Let's build a vector that allows you to search against the `title` and `body` fields of the `Post` model.

Run the following code in the Python shell:

```
>>> from django.contrib.postgres.search import SearchVector
>>> from blog.models import Post
>>>
>>> Post.objects.annotate(
...     search=SearchVector('title', 'body'),
... ).filter(search='django')
<QuerySet [<Post: Markdown post>, <Post: Who was Django Reinhardt?>]>
```

Using `annotate` and defining `SearchVector` with both fields, you provide a functionality to match the query against both the `title` and `body` of the posts.

Full-text search is an intensive process. If you are searching for more than a few hundred rows, you should define a functional index that matches the search vector you are using. Django provides a `SearchVectorField` field for your models. You can read more about this at `https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/#performance`.

Building a search view

Now, you will create a custom view to allow your users to search posts. First, you will need a search form. Edit the `forms.py` file of the `blog` application and add the following form:

```
class SearchForm(forms.Form):
    query = forms.CharField()
```

You will use the `query` field to let users introduce search terms. Edit the `views.py` file of the `blog` application and add the following code to it:

```
# ...
from django.contrib.postgres.search import SearchVector
from .forms import EmailPostForm, CommentForm, SearchForm

# ...

def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            results = Post.published.annotate(
                search=SearchVector('title', 'body'),
            ).filter(search=query)

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

In the preceding view, first, we instantiate the `SearchForm` form. To check whether the form is submitted, we look for the `query` parameter in the `request.GET` dictionary. We send the form using the `GET` method instead of `POST` so that the resulting URL includes the `query` parameter and is easy to share. When the form is submitted, we instantiate it with the submitted `GET` data, and verify that the form data is valid. If the form is valid, we search for published posts with a custom `SearchVector` instance built with the `title` and `body` fields.

The search view is now ready. We need to create a template to display the form and the results when the user performs a search.

Create a new file inside the `templates/blog/post/` directory, name it `search.html`, and add the following code to it:

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

```
{% block title %}Search{% endblock %}

{% block content %}
  {% if query %}
    <h1>Posts containing "{{ query }}"</h1>
    <h3>
      {% with results.count as total_results %}
        Found {{ total_results }} result{{ total_results|pluralize }}
      {% endwith %}
    </h3>
    {% for post in results %}
      <h4>
        <a href="{{ post.get_absolute_url }}">
          {{ post.title }}
        </a>
      </h4>
      {{ post.body|markdown|truncatewords_html:12 }}
    {% empty %}
      <p>There are no results for your query.</p>
    {% endfor %}
    <p><a href="{% url "blog:post_search" %}">Search again</a></p>
  {% else %}
    <h1>Search for posts</h1>
    <form method="get">
      {{ form.as_p }}
      <input type="submit" value="Search">
    </form>
  {% endif %}
{% endblock %}
```

As in the search view, we distinguish whether the form has been submitted by the presence of the `query` parameter. Before the query is submitted, we display the form and a submit button. When the search form is submitted, we display the query performed, the total number of results, and the list of posts that match the search query.

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

```
urlpatterns = [
    # Post views
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
```

```
    path('tag/<slug:tag_slug>/',
         views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
    path('feed/', LatestPostsFeed(), name='post_feed'),
    path('search/', views.post_search, name='post_search'),
]
```

Next, open `http://127.0.0.1:8000/blog/search/` in your browser. You should see the following search form:

Figure 3.27: The form with the query field to search for posts

Enter a query and click on the **SEARCH** button. You will see the results of the search query, as follows:

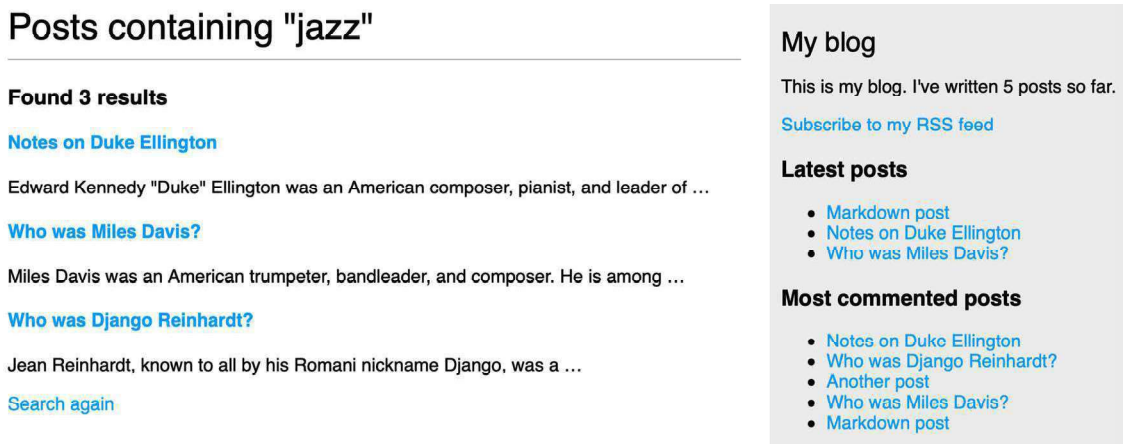

Figure 3.28: Search results for the term "jazz"

Congratulations! You have created a basic search engine for your blog.

Stemming and ranking results

Stemming is the process of reducing words to their word stem, base, or root form. Stemming is used by search engines to reduce indexed words to their stem, and to be able to match inflected or derived words. For example, the words "music", "musical" and "musicality" can be considered similar words by a search engine. The stemming process normalizes each search token into a lexeme, a unit of lexical meaning that underlies a set of words that are related through inflection. The words "music", "musical" and "musicality" would convert to "music" when creating a search query.

Django provides a `SearchQuery` class to translate terms into a search query object. By default, the terms are passed through stemming algorithms, which helps you to obtain better matches.

The PostgreSQL search engine also removes stop words, such as "a", "the", "on", and "of". Stop words are a set of commonly used words in a language. They are removed when creating a search query because they appear too frequently to be relevant to searches. You can find the list of stop words used by PostgreSQL for the English language at `https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/english.stop`.

We also want to order results by relevancy. PostgreSQL provides a ranking function that orders results based on how often the query terms appear and how close together they are.

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

```
from django.contrib.postgres.search import SearchVector, \
                                           SearchQuery, SearchRank
```

Then, edit the `post_search` view, as follows. New code is highlighted in bold:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            search_vector = SearchVector('title', 'body')
            search_query = SearchQuery(query)
            results = Post.published.annotate(
                search=search_vector,
                rank=SearchRank(search_vector, search_query)
            ).filter(search=search_query).order_by('-rank')

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

In the preceding code, we create a `SearchQuery` object, filter results by it, and use `SearchRank` to order the results by relevancy.

You can open `http://127.0.0.1:8000/blog/search/` in your browser and test different searches to test stemming and ranking. The following is an example of ranking by the number of occurrences of the word `django` in the title and body of the posts:

Figure 3.29: Search results for the term "django"

Stemming and removing stop words in different languages

We can set up `SearchVector` and `SearchQuery` to execute stemming and remove stop words in any language. We can pass a `config` attribute to `SearchVector` and `SearchQuery` to use a different search configuration. This allows us to use different language parsers and dictionaries. The following example executes stemming and removes stops in Spanish:

```
search_vector = SearchVector('title', 'body', config='spanish')
search_query = SearchQuery(query, config='spanish')
results = Post.published.annotate(
    search=search_vector,
    rank=SearchRank(search_vector, search_query)
).filter(search=search_query).order_by('-rank')
```

You can find the Spanish stop words dictionary used by PostgreSQL at `https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/spanish.stop`.

Weighting queries

We can boost specific vectors so that more weight is attributed to them when ordering results by relevancy. For example, we can use this to give more relevance to posts that are matched by title rather than by content.

Edit the `views.py` file of the `blog` application and modify the `post_search` view as follows. New code is highlighted in bold:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            search_vector = SearchVector('title', weight='A') + \
                            SearchVector('body', weight='B')
            search_query = SearchQuery(query)
            results = Post.published.annotate(
                search=search_vector,
                rank=SearchRank(search_vector, search_query)
            ).filter(rank__gte=0.3).order_by('-rank')
```

```
    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

In the preceding code, we apply different weights to the search vectors built using the `title` and `body` fields. The default weights are `D`, `C`, `B`, and `A`, and they refer to the numbers `0.1`, `0.2`, `0.4`, and `1.0`, respectively. We apply a weight of `1.0` to the `title` search vector (`A`) and a weight of `0.4` to the `body` vector (`B`). Title matches will prevail over body content matches. We filter the results to display only the ones with a rank higher than `0.3`.

Searching with trigram similarity

Another search approach is trigram similarity. A trigram is a group of three consecutive characters. You can measure the similarity of two strings by counting the number of trigrams that they share. This approach turns out to be very effective for measuring the similarity of words in many languages.

To use trigrams in PostgreSQL, you will need to install the `pg_trgm` extension first. Execute the following command in the shell prompt to connect to your database:

```
psql blog
```

Then, execute the following command to install the `pg_trgm` extension:

```
CREATE EXTENSION pg_trgm;
```

You will get the following output:

```
CREATE EXTENSION
```

Let's edit the view and modify it to search for trigrams.

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

```
from django.contrib.postgres.search import TrigramSimilarity
```

Then, modify the `post_search` view as follows. New code is highlighted in bold:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
```

```
            query = form.cleaned_data['query']
            results = Post.published.annotate(
                similarity=TrigramSimilarity('title', query),
            ).filter(similarity__gt=0.1).order_by('-similarity')

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

Open `http://127.0.0.1:8000/blog/search/` in your browser and test different searches for trigrams. The following example displays a hypothetical typo in the `django` term, showing search results for `yango`:

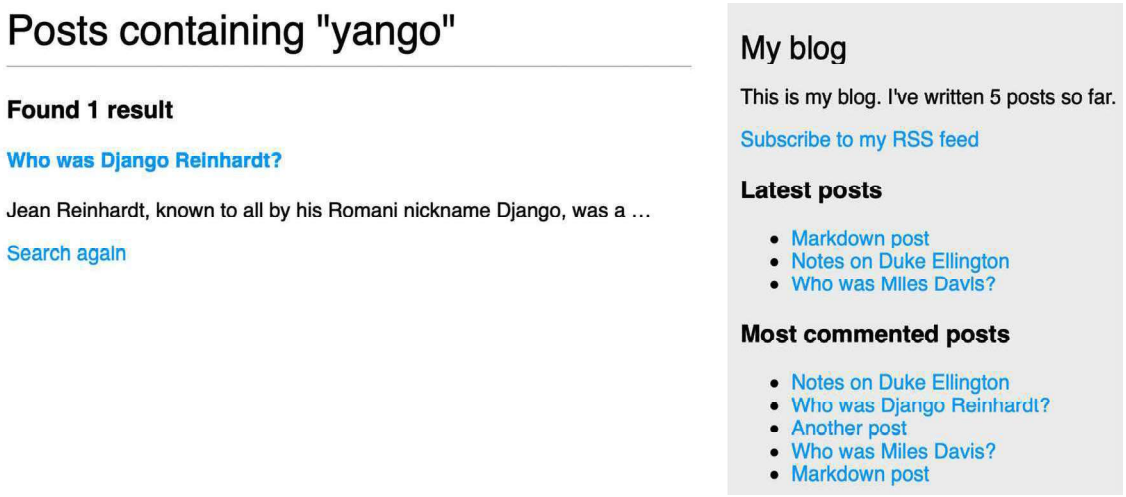

Figure 3.30: Search results for the term "yango"

We have added a powerful search engine to the blog application.

You can find more information about full-text search at `https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/`.

Additional resources

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

- Source code for this chapter – `https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03`
- Django-taggit – `https://github.com/jazzband/django-taggit`

- Django-taggit ORM managers – `https://django-taggit.readthedocs.io/en/latest/api.html`
- Many-to-many relationships – `https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/`
- Django aggregation functions – `https://docs.djangoproject.com/en/4.1/topics/db/aggregation/`
- Built-in template tags and filters – `https://docs.djangoproject.com/en/4.1/ref/templates/builtins/`
- Writing custom template tags – `https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/`
- Markdown format reference – `https://daringfireball.net/projects/markdown/basics`
- Django Sitemap framework – `https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/`
- Django Sites framework – `https://docs.djangoproject.com/en/4.1/ref/contrib/sites/`
- Django syndication feed framework – `https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/`
- PostgreSQL downloads – `https://www.postgresql.org/download/`
- PostgreSQL full-text search capabilities – `https://www.postgresql.org/docs/14/textsearch.html`
- Django support for PostgreSQL full-text search – `https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/`

Summary

In this chapter, you implemented a tagging system by integrating a third-party application with your project. You generated post recommendations using complex QuerySets. You also learned how to create custom Django template tags and filters to provide templates with custom functionalities. You also created a sitemap for search engines to crawl your site and an RSS feed for users to subscribe to your blog. You then built a search engine for your blog using the full-text search engine of PostgreSQL.

In the next chapter, you will learn how to build a social website using the Django authentication framework and how to implement user account functionalities and custom user profiles.