

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

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

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

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

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

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

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

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

```
<input type='hidden' name='csrfmiddlewaretoken' value='26JjKo2lcEtYkGo
V9z4XmJIEHLXN5LDR' />
```



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

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

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

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

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

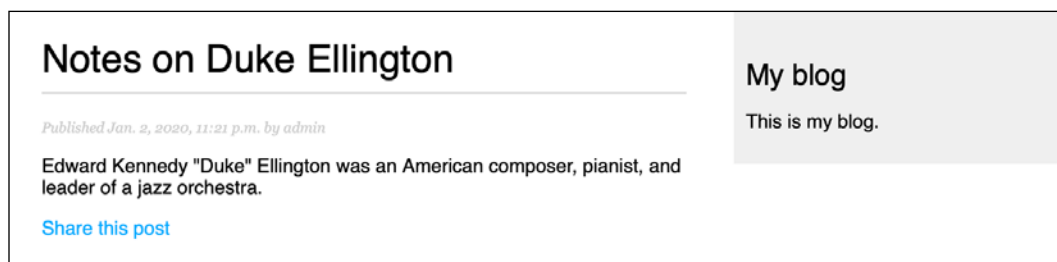
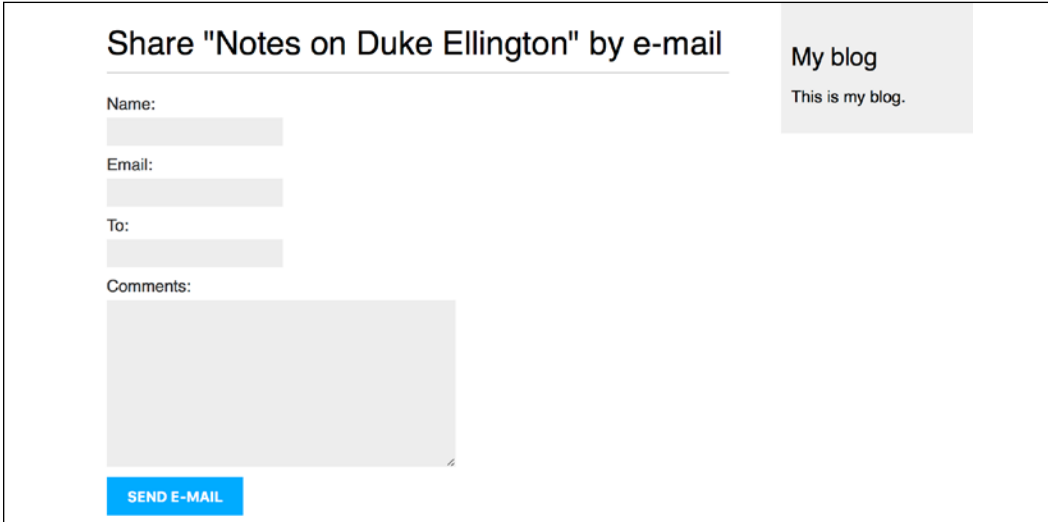


Figure 2.2: The post detail page, including a link to share the post

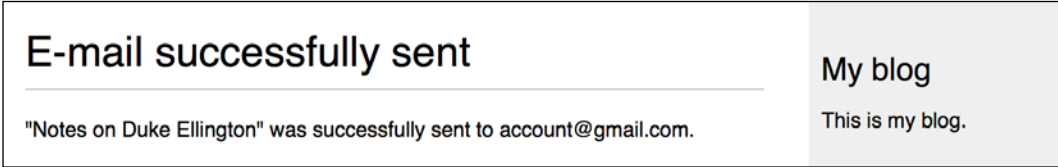
Click on **Share this post**, and you should see the page, including the form to share this post by email, as follows:



The screenshot shows a web form titled "Share 'Notes on Duke Ellington' by e-mail". On the right side, there is a grey sidebar with the text "My blog" and "This is my blog." below it. The form itself has four input fields: "Name:", "Email:", "To:", and "Comments:". The "Comments:" field is a larger text area. At the bottom left of the form is a blue button labeled "SEND E-MAIL".

Figure 2.3: The page to share a post via email

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



The screenshot shows a success message page. The title is "E-mail successfully sent". Below the title, it says "Notes on Duke Ellington" was successfully sent to account@gmail.com. On the right side, there is a grey sidebar with the text "My blog" and "This is my blog." below it.

Figure 2.4: A success message for a post shared via email

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

The screenshot shows a web form titled "Share 'Notes on Duke Ellington' by e-mail". The form is divided into two main sections. On the right, a grey sidebar contains the text "My blog" and "This is my blog.". The main form area has the following fields and errors:

- Name:** A text input field containing "Antonio".
- Email:** A text input field containing "invalid". Above this field is a red error message: "• Enter a valid email address."
- To:** An empty text input field. Below it is a red error message: "• This field is required."
- Comments:** A large, empty text area.
- SEND E-MAIL:** A blue button at the bottom of the form.

Figure 2.5: The share post form displaying invalid data errors

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

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

Creating a comment system

You will build a comment system wherein users will be able to comment on posts. To build the comment system, you need to do the following:

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

Building a model

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

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

    class Meta:
        ordering = ('created',)

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

This is your `Comment` model. It contains a `ForeignKey` to associate a comment with a single post. This many-to-one relationship is defined in the `Comment` model because each comment will be made on one post, and each post may have multiple comments.

The `related_name` attribute allows you to name the attribute that you use for the relationship from the related object back to this one. After defining this, you can retrieve the post of a comment object using `comment.post` and retrieve all comments of a post using `post.comments.all()`. If you don't define the `related_name` attribute, Django will use the name of the model in lowercase, followed by `_set` (that is, `comment_set`) to name the relationship of the related object to the object of the model, where this relationship has been defined.

You can learn more about many-to-one relationships at https://docs.djangoproject.com/en/3.0/topics/db/examples/many_to_one/.

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

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

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

You should see the following output:

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

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

```
python manage.py migrate
```

You will get an output that includes the following line:

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

The migration that you just created has been applied; now a `blog_comment` table exists in the database.

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

```
from .models import Post, Comment

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

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

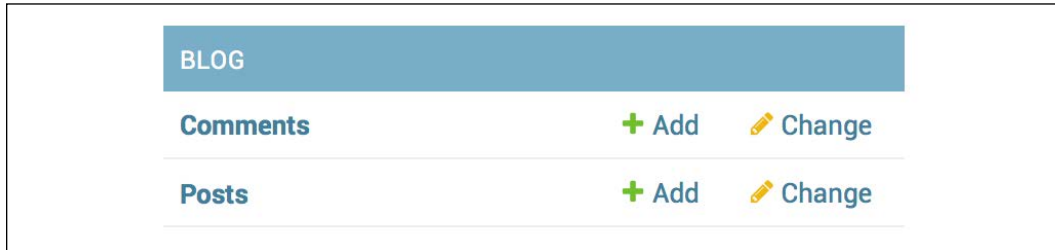


Figure 2.6: Blog application models on the Django administration index page

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

Creating forms from models

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

```
from .models import Comment

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

To create a form from a model, you just need to indicate which model to use to build the form in the `Meta` class of the form. Django introspects the model and builds the form dynamically for you.

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

Handling ModelForms in views

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

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

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

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

    new_comment = None

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

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

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


You build this `QuerySet`, starting from the `post` object. Instead of building a `QuerySet` for the `Comment` model directly, you leverage the `post` object to retrieve the related `Comment` objects. You use the manager for the related objects that you defined as `comments` using the `related_name` attribute of the relationship in the `Comment` model. You use the same view to let your users add a new comment. You initialize the `new_comment` variable by setting it to `None`. You will use this variable when a new comment is created.

You build a form instance with `comment_form = CommentForm()` if the view is called by a `GET` request. If the request is done via `POST`, you instantiate the form using the submitted data and validate it using the `is_valid()` method. If the form is invalid, you render the template with the validation errors. If the form is valid, you take the following actions:

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

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

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



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

2. You assign the current post to the comment you just created:

```
new_comment.post = post
```

By doing this, you specify that the new comment belongs to this post.

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

```
new_comment.save()
```

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

Adding comments to the post detail template

You have created the functionality to manage comments for a post. Now you need to adapt your `post/detail.html` template to do the following things:

- Display the total number of comments for a post

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

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

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

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



The `{% with %}` template tag is useful for avoiding hitting the database or accessing expensive methods multiple times.

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

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

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

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

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

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

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

The code is pretty straightforward: if the `new_comment` object exists, you display a success message because the comment was successfully created. Otherwise, you render the form with a paragraph, `<p>`, element for each field and include the CSRF token required for POST requests.

Open `http://127.0.0.1:8000/blog/` in your browser and click on a post title to take a look at its detail page. You will see something like the following screenshot:

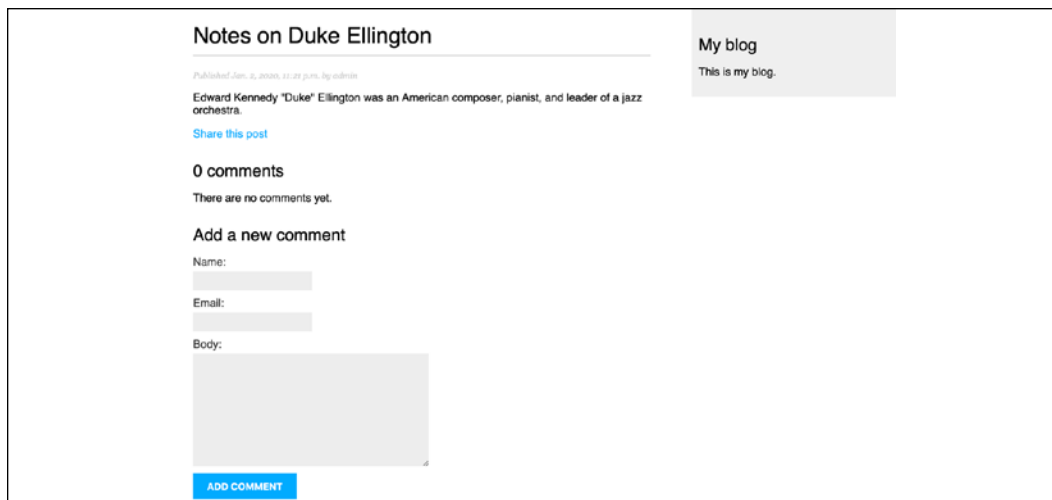


Figure 2.7: The post detail page, including the form to add a comment

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

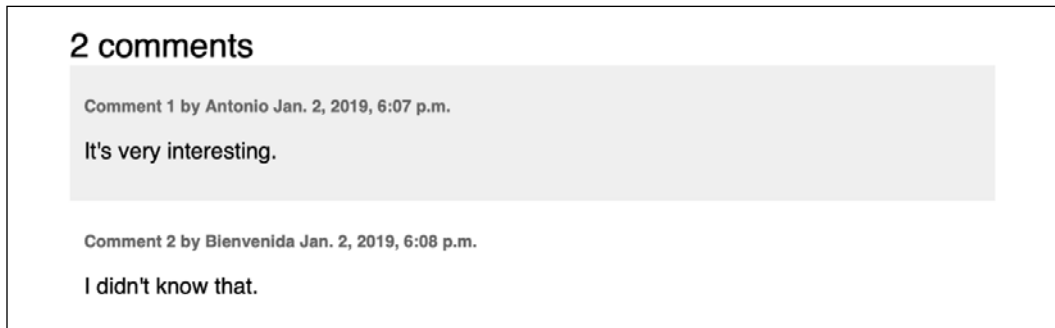


Figure 2.8: The comment list on the post detail page

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

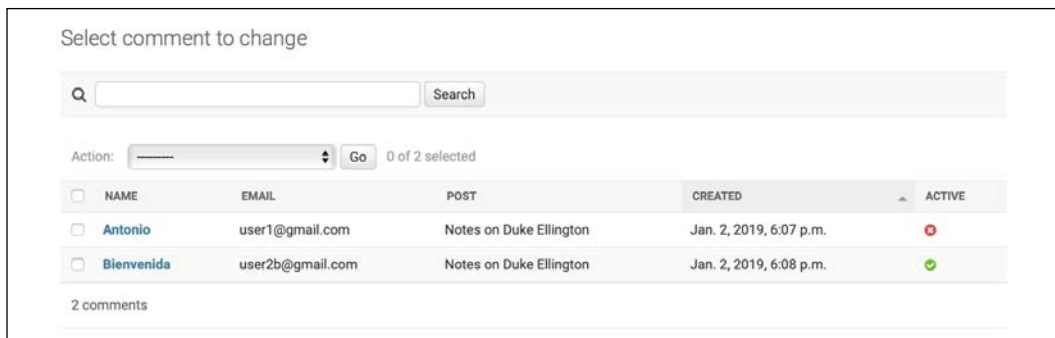


Figure 2.9: Active/inactive comments on the Django administration site

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

Adding the tagging functionality

After implementing your comment system, you need to create a way to tag your posts. You will do this by integrating a third-party Django tagging application into your 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==1.2.0
```

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

```
INSTALLED_APPS = [  
    # ...  
    '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.

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

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

You should get the following output:

```
Migrations for 'blog':  
  blog/migrations/0003_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 blog.0003_post_tags... OK
```

Your database is now ready to use `django-taggit` models.

Let's explore how to use the `tags` manager. Open the terminal with the `python manage.py shell` command and enter the following code. First, you will retrieve one of your 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>]>
```

That was easy, right? Run the `python manage.py runserver` command to start the development server again and 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:

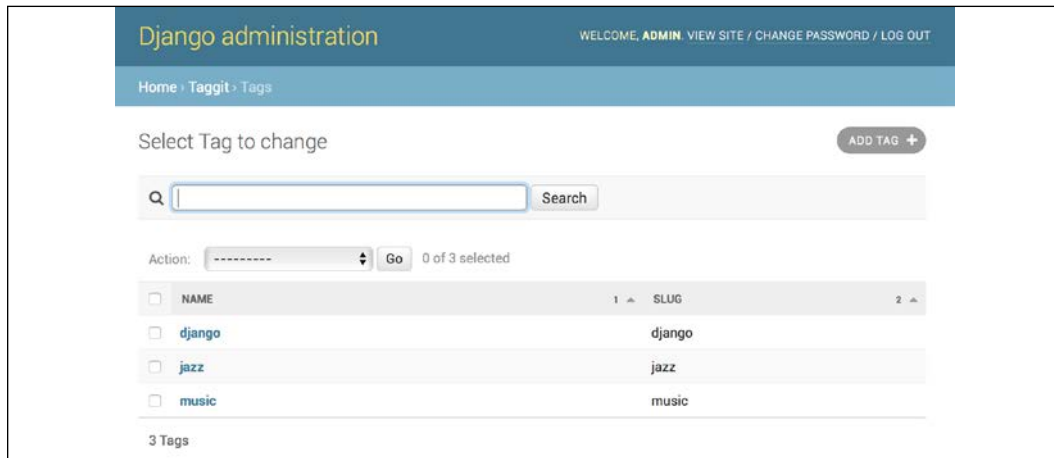


Figure 2.10: The tag change list view on the Django administration site

Navigate to `http://127.0.0.1:8000/admin/blog/post/` and click on a post to edit it. You will see that posts now include a new **Tags** field, as follows, where you can easily edit tags:

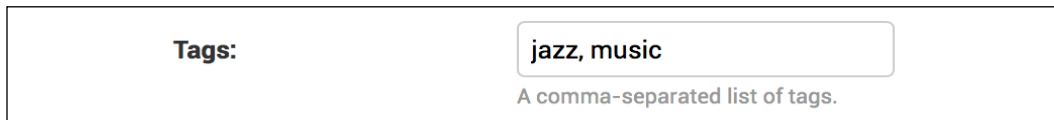


Figure 2.11: The related tags field of a Post object

Now, you need to edit your blog posts to display tags. Open the `blog/post/list.html` template and add the following HTML code below the post title:

```
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
```

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:

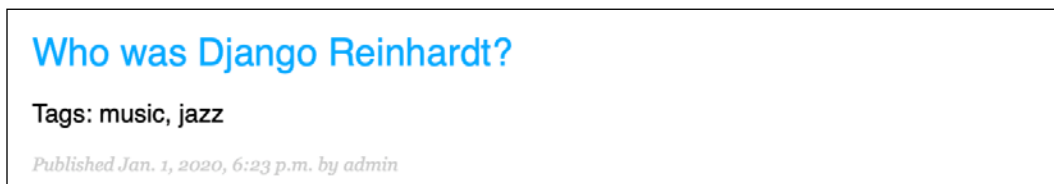


Figure 2.12: The Post list item, including related tags

Next, you 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:

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    object_list = Post.published.all()
    tag = None

    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list = object_list.filter(tags__in=[tag])

    paginator = Paginator(object_list, 3) # 3 posts in each page
    # ...
```

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, you build the initial `QuerySet`, retrieving all published posts, and if there is a given tag slug, you get the `Tag` object with the given slug using the `get_object_or_404()` shortcut.
3. Then, you filter the list of posts by the ones that contain the given tag. Since this is a many-to-many relationship, you have to filter posts by tags contained in a given list, which, in your case, contains only one element. You 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 your 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 5, Sharing Content on Your Website*. You can discover more about many-to-many relationships at https://docs.djangoproject.com/en/3.0/topics/db/examples/many_to_many/.

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.

Finally, modify the `render()` function at the bottom of the view to pass the `tag` variable to the template. The view should look like this:

```
def post_list(request, tag_slug=None):
    object_list = Post.published.all()
    tag = None
```



```
if tag_slug:
    tag = get_object_or_404(Tag, slug=tag_slug)
    object_list = object_list.filter(tags__in=[tag])

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

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 you are naming them differently. 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.

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 above the `{% for %}` loop:

```
{% if tag %}
    <h2>Posts tagged with "{{ tag.name }}"</h2>
{% endif %}
```

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, change the way tags are displayed, as follows:

```
<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>
```

In the code above, you loop through all the tags of a post displaying a custom link to the URL to filter posts by that tag. You 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/` in your browser and click on any tag link. You will see the list of posts filtered by that tag, like this:

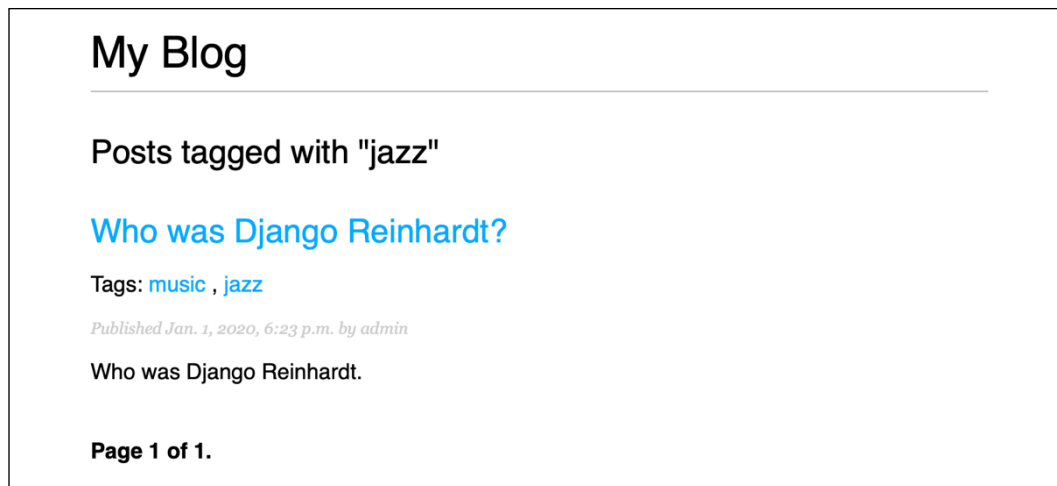


Figure 2.13: A post filtered by the tag "jazz"

Retrieving posts by similarity

Now that you have implemented tagging for your 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. You will build a functionality to display similar posts by the number of tags they share. In this way, when a user reads a post, you 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. In 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/3.0/topics/db/aggregation/>.

Add the following lines inside the `post_detail` view before the `render()` function, with the same indentation level:

```
# 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]

```

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.

Add the `similar_posts` object to the context dictionary for the `render()` function, as follows:

```

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

```

Now, edit the `blog/post/detail.html` template and add the following code before the post comment list:

```

<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 %}

```

The post detail page should look like this:

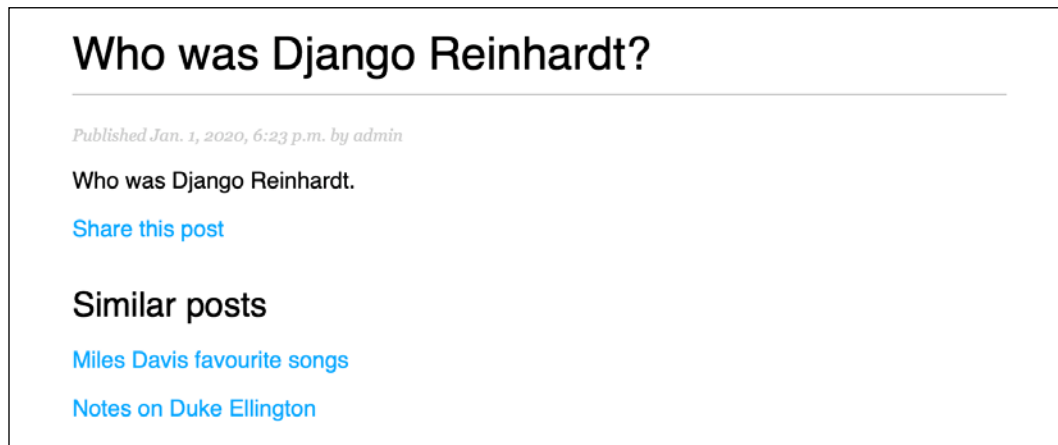


Figure 2.14: The post detail page, including a list of similar posts

You are now able to successfully recommend similar posts to your users. `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.

Summary

In this chapter, you learned how to work with Django forms and model forms. You created a system to share your site's content by email and created a comment system for your blog. You added tagging to your blog posts, integrating a reusable application, and built complex QuerySets to retrieve objects by similarity.

In the next chapter, you will learn how to create custom template tags and filters. You will also build a custom sitemap and feed for your blog posts, and implement the full text search functionality for your posts.

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, and you implemented a tagging system by integrating a third-party application with your project. In this chapter, you will extend your blog application with some other popular features used on blogging platforms. You will also learn about other components and functionalities with Django.

The chapter will cover the following points:

- **Creating custom template tags and filters:** You will learn how to build your own template tags and template filters to exploit the capabilities of Django templates.
- **Adding a sitemap and post feed:** You will learn how to use the sitemaps framework and syndication framework that come with Django.
- **Implementing full-text search with PostgreSQL:** Search is a very popular feature for blogs. You will learn how to implement an advanced search engine for your blog application.

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/3.0/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 could be a tag to perform a QuerySet or any server-side processing that you want to reuse across templates. For example, you could build a template tag to display the list of latest posts published on your blog. You can include this list in the sidebar of the blog for multiple pages, regardless of the view.

Custom template tags

Django provides the following helper functions that allow you to create your own template tags in an easy manner:

- `simple_tag`: Processes the data and returns a string
- `inclusion_tag`: Processes the 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.

Let's start by creating a simple tag to retrieve the total posts published on the blog. Edit the `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()
```

You have created a simple template tag that returns the number of posts published so far. 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 your own template tags and filters.

In the code above, you define a tag called `total_posts` with a Python function and use the `@register.simple_tag` decorator to register the function 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, you have to make them available for the template using the `{% load %}` tag. As mentioned before, you need to use the name of the Python module containing your template tags and filters.

Open 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. Just add `{% total_posts %}` to your template. The template should look like this:

```
{% 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:

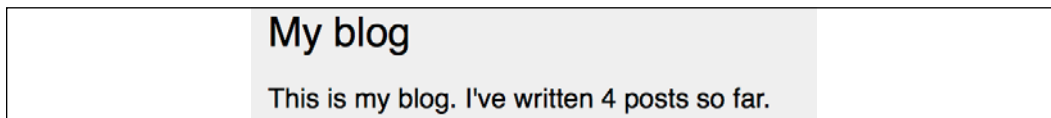


Figure 3.1: The total posts published included in the sidebar

The power of custom template tags is that you can 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.

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

Edit the `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, you register the template tag using `@register.inclusion_tag` and specify the template that will be rendered with the returned values using `blog/post/latest_posts.html`. Your template tag will accept an optional `count` parameter that defaults to 5. This parameter you to specify the number of posts that you want to display. You 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 you just created allows you 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`. 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. The sidebar code should look like the following:

```
<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>
```

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 3.2: The sidebar, including the latest published posts

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

Edit the `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 field `total_comments` 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/3.0/topics/db/aggregation/>.

Next, edit the `blog/base.html` template and append the following code to the sidebar `<div>` element:

```
<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>
```

In the preceding code, you store the result in a custom variable using the `as` argument followed by the variable name. For your template tag, you 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, you display the returned posts using an unordered list.

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

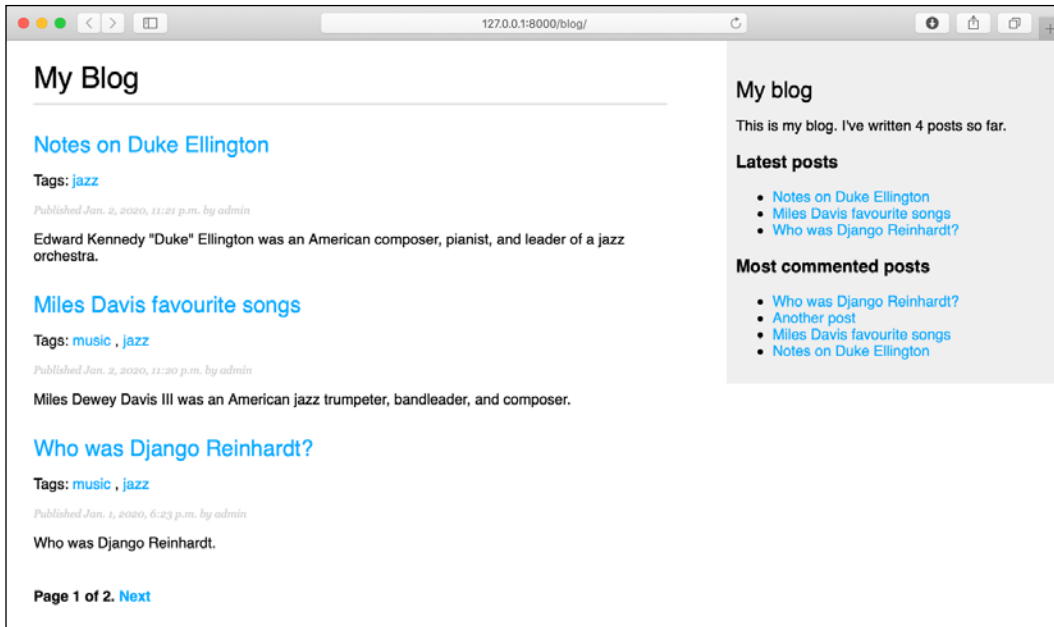


Figure 3.3: The post list view, including the complete sidebar with the latest and most commented posts

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/3.0/howto/custom-template-tags/>.

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 looks like `{{ variable|my_filter }}`. Filters with an argument look 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 of them 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/3.0/ref/templates/builtins/#built-in-filter-reference>.

You will create a custom filter to enable you to use markdown syntax in your blog posts and then convert the post contents 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:

```
pip install markdown==3.2.1
```

Then, edit the `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))
```

You register template filters in the same way as template tags. To prevent a name clash between your function name and the markdown module, you name your function `markdown_format` and name 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 `<p>`; (*less than* symbol, *p* character, *greater than* symbol). You 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.

Now, load your template tags module in the post list and detail templates. Add the following line at the top of the `blog/post/list.html` and `blog/post/detail.html` templates after the `{% extends %}` tag:

```
{% load blog_tags %}
```

In the `post/detail.html` template, look for the following line:

```
{{ post.body|linebreaks }}
```

Replace it with the following one:

```
{{ post.body|markdown }}
```

Then, in the `post/list.html` template, find the following line:

```
{{ post.body|truncatewords:30|linebreaks }}
```

Replace it with the following one:

```
{{ post.body|markdown|truncatewords_html:30 }}
```

The `truncatewords_html` 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 add a 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/>)

Open your browser and take a look at how the post is rendered. You should see the following output:

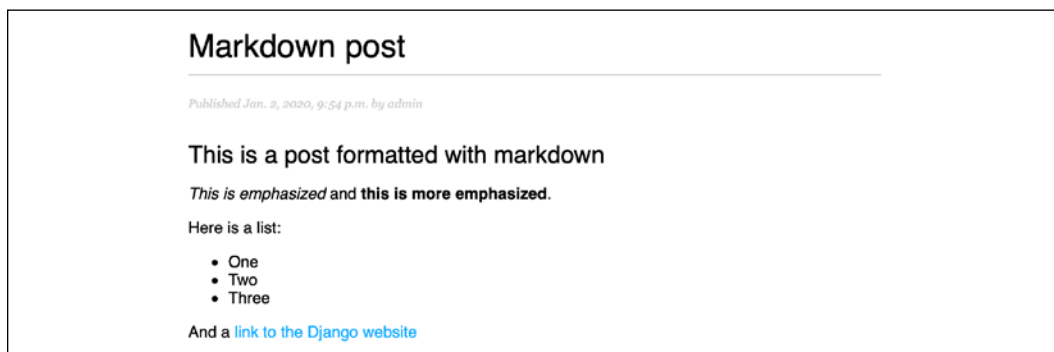


Figure 3.4: The post with markdown content rendered as HTML

As you can see in the preceding screenshot, custom template filters are very useful for customizing formatting. You can find more information about custom filters at <https://docs.djangoproject.com/en/3.0/howto/custom-template-tags/#writing-custom-template-filters>.

Adding a sitemap to your 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: sitemaps help 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, you will need to activate both the `sites` and the `sitemap` applications in your project.

Edit the `settings.py` file of your 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:

```
SITE_ID = 1

# Application definition
INSTALLED_APPS = [
    # ...
    'django.contrib.sites',
    'django.contrib.sitemaps',
]
```

Now run the following command 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

```

You create 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 you created this method in *Chapter 1, Building a Blog Application*, to retrieve 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/3.0/ref/contrib/sitemaps/>.

Finally, you just need to add your sitemap URL. Edit the main `urls.py` file of your project and add the sitemap, as follows:

```

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, you include the required imports and define a dictionary of sitemaps. You define a URL pattern that matches `sitemap.xml` and uses the `sitemap` view. The `sitemaps` dictionary is passed to the `sitemap` view.

Now run the development server and open `http://127.0.0.1:8000/sitemap.xml` in your browser. You will see the following XML output:

```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://example.com/blog/2020/1/2/markdown-post/</loc>
    <lastmod>2020-01-02</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>
http://example.com/blog/2020/1/1/who-was-django-reinhardt/
</loc>
    <lastmod>2020-01-02</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

The URL for each post has been built 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.

You can see that 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.

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

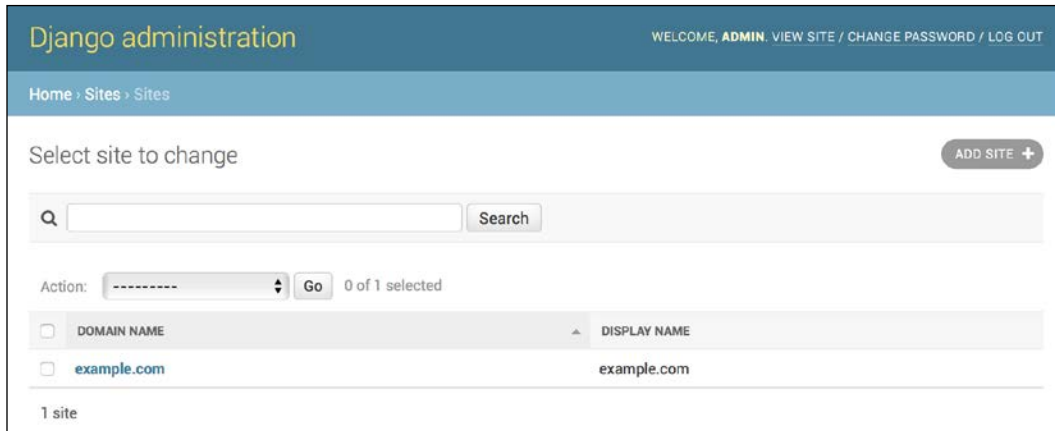


Figure 3.5: The Django administration list view for the Site model of the site's framework

The preceding screenshot 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. In order to generate URLs that exist in your local environment, change the domain name to `localhost:8000`, as shown in the following screenshot, and save it:

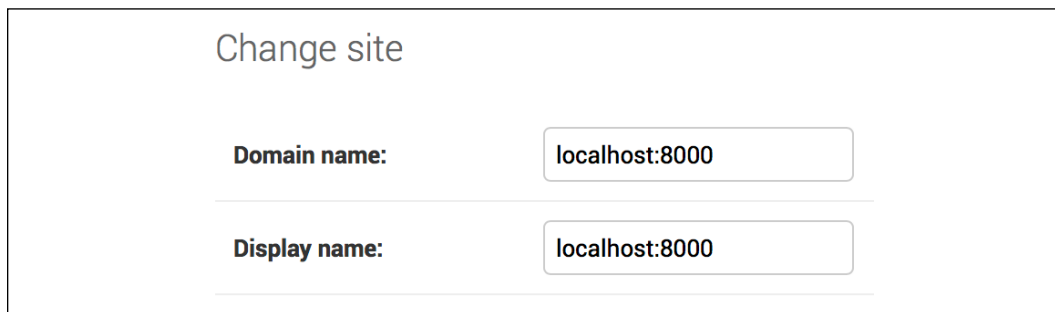


Figure 3.6: The Django administration edit view for the Site model of the site's framework

The URLs displayed in your feed will now be built using this hostname. In a production environment, you will have to use your own domain name for the site's framework.

Creating feeds for your 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 will be able to subscribe to your feed using a feed aggregator (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:

```
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords
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(item.body, 30)
```

First, you subclass the `Feed` class of the syndication framework. The `title`, `link`, and `description` attributes correspond to the `<title>`, `<link>`, and `<description>` RSS elements, respectively.

You 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. You used `reverse()` in *Chapter 1, Building a Blog Application*. 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. You are retrieving only the last five published posts for this feed. The `item_title()` and `item_description()` methods will receive each object returned by `items()` and return the title and description for each item. You use the `truncatewords` built-in template filter to build the description of the blog post with the first 30 words.

Now edit the `blog/urls.py` file, import the `LatestPostsFeed` you just created, and instantiate the feed in a new URL pattern:

```
from .feeds import LatestPostsFeed

urlpatterns = [
    # ...
    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 open the same URL in an RSS client, you will be able to see your feed with a user-friendly interface.

The final step is to add a feed subscription link to the blog's sidebar. Open the `blog/base.html` template and add the following line under the number of total posts inside the sidebar div:

```
<p>
  <a href="{% url 'blog:post_feed' %}">Subscribe to my RSS feed</a>
</p>
```

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



Figure 3.7: 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/3.0/ref/contrib/syndication/>.

Adding full-text search to your blog

Next, you will add search capabilities to your 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. 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 full-text search at <https://www.postgresql.org/docs/12/static/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

You are currently using SQLite for your `blog` project. This is sufficient for development purposes. However, for a production environment, you will need a more powerful database, such as PostgreSQL, MariaDB, MySQL, or Oracle. You will change your database to PostgreSQL to benefit from its full-text search features.

If you are using Linux, install PostgreSQL with the following command:

```
sudo apt-get install postgresql postgresql-contrib
```

If you are using macOS or Windows, download PostgreSQL from <https://www.postgresql.org/download/> and install it.

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

```
pip install psycopg2-binary==2.8.4
```

Let's create a user for your PostgreSQL database. Open the shell and run the following commands:

```
su postgres  
createuser -dP blog
```

You will be prompted for a password for the new user. Enter the desired password and then create the `blog` database and give ownership to the `blog` user you just created with the following command:

```
createdb -E utf8 -U blog blog
```

Then, edit the `settings.py` file of your project and modify the `DATABASES` setting to make it look as follows:

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

```
}
```

Replace the preceding data with the database name and credentials for the user you created. The new database is empty. Run the following command to apply all database migrations:

```
python manage.py migrate
```

Finally, create a superuser with the following command:

```
python manage.py createsuperuser
```

You can now run the development server and access the administration site at `http://127.0.0.1:8000/admin/` with the new superuser.

Since you switched the database, there are no posts stored in it. Populate your new database with a couple of sample blog posts so that you can perform searches against the database.

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.postgres',  
]
```

Now you can search against a single field using the `search QuerySet` lookup, like this:

```
from blog.models import Post  
Post.objects.filter(body__search='django')
```

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:

```

from django.contrib.postgres.search import SearchVector
from blog.models import Post

Post.objects.annotate(
    search=SearchVector('title', 'body'),
).filter(search='django')

```

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/3.0/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, you instantiate the `SearchForm` form. To check whether the form is submitted, you look for the `query` parameter in the `request.GET` dictionary. You 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, you instantiate it with the submitted `GET` data, and verify that the form data is valid. If the form is valid, you search for published posts with a custom `SearchVector` instance built with the `title` and `body` fields.

The search view is ready now. You need to create a template to display the form and the results when the user performs a search. Create a new file inside the `blog/post/` template 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:5 }}
            {% 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, you can distinguish whether the form has been submitted by the presence of the `query` parameter. Before the query is submitted, you display the form and a submit button. After the post is submitted, you display the query performed, the total number of results, and the list of posts returned.

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

```
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:



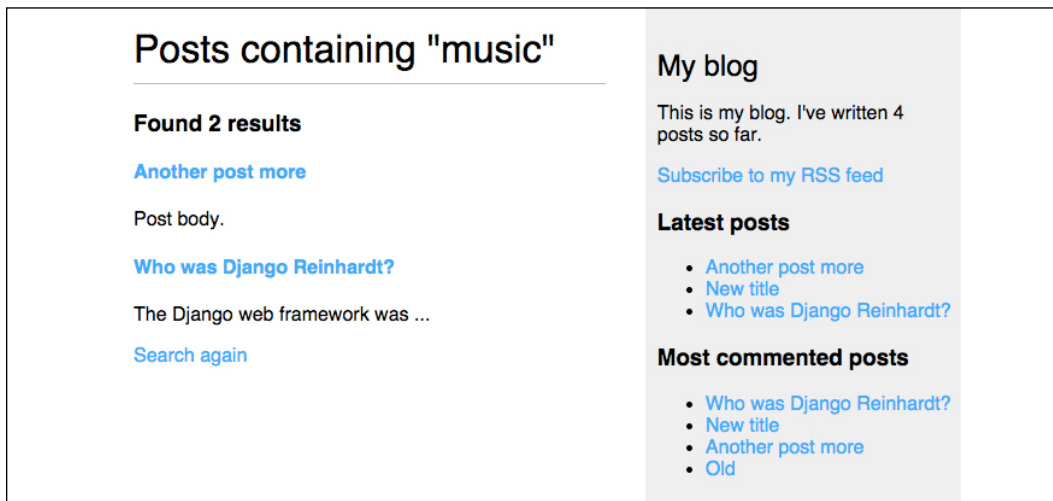
Search for posts

Query:

SEARCH

Figure 3.8: 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:



Posts containing "music"

Found 2 results

[Another post more](#)

Post body.

[Who was Django Reinhardt?](#)

The Django web framework was ...

[Search again](#)

My blog

This is my blog. I've written 4 posts so far.

[Subscribe to my RSS feed](#)

Latest posts

- [Another post more](#)
- [New title](#)
- [Who was Django Reinhardt?](#)

Most commented posts

- [Who was Django Reinhardt?](#)
- [New title](#)
- [Another post more](#)
- [Old](#)

Figure 3.9: Search results for the term "music"

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, "music" and "musician" can be considered similar words by a search engine.

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. You 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 your blog application and add the following imports:

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

Then, take a look at the following lines:

```
results = Post.published.annotate(  
    search=SearchVector('title', 'body'),  
).filter(search=query)
```

Replace them with the following ones:

```
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')
```

In the preceding code, you 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 for the word `django` in the title and body of the posts:

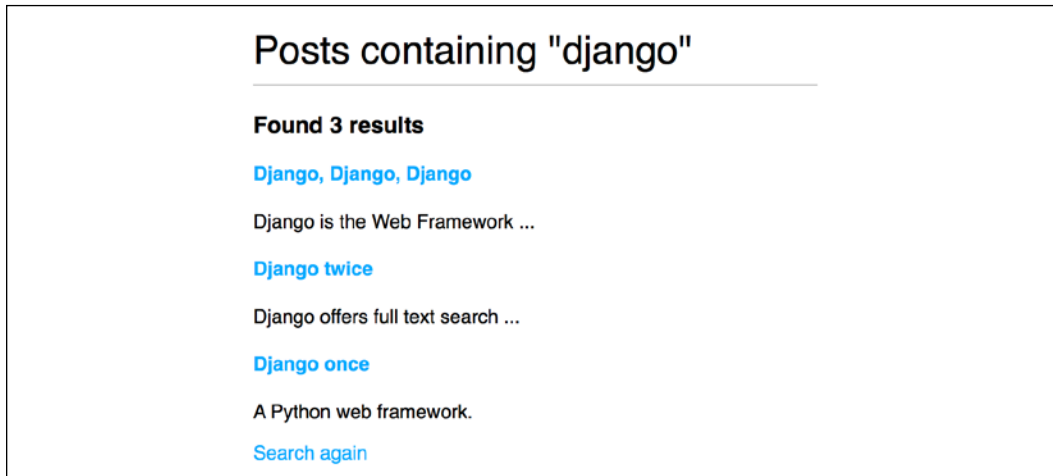


Figure 3.10: Search results for the term "django"

Weighting queries

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

Edit the previous lines of the `views.py` file of your `blog` application and make them look like this:

```
search_vector = SearchVector('title', weight='A') + \
                SearchVector('body', weight='B')
search_query = SearchQuery(query)
results = Post.published.annotate(
    rank=SearchRank(search_vector, search_query)
).filter(rank_gte=0.3).order_by('-rank')
```

In the preceding code, you 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. You apply a weight of `1.0` to the `title` search vector and a weight of `0.4` to the `body` vector. Title matches will prevail over body content matches. You 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.

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

```
psql blog
```

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

```
CREATE EXTENSION pg_trgm;
```

Let's edit your 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, replace the `Post` search query with the following lines:

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

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`:

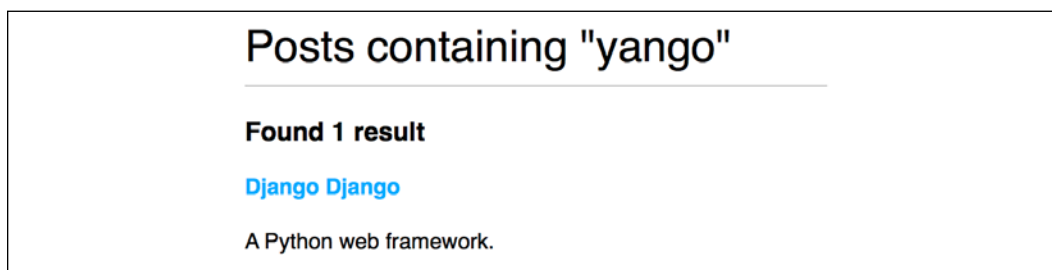


Figure 3.11: Search results for the term "yango"

Now you have a powerful search engine built into your project. You can find more information about full-text search at <https://docs.djangoproject.com/en/3.0/ref/contrib/postgres/search/>.

Other full-text search engines

You may want to use a full-text search engine other than from PostgreSQL. If you want to use Solr or Elasticsearch, you can integrate them into your Django project using Haystack. Haystack is a Django application that works as an abstraction layer for multiple search engines. It offers a simple search API that is very similar to Django QuerySets. You can find more information about Haystack at <https://django-haystack.readthedocs.io/en/master/>.

Summary

In this chapter, you learned how to create custom Django template tags and filters to provide templates with a custom functionality. 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, create custom user profiles, and build social authentication.

4

Building a Social Website

In the preceding chapter, you learned how to create sitemaps and feeds, and you built a search engine for your blog application. In this chapter, you will discover how to develop a social application, which means that users are able to join an online platform and interact with each other by sharing content. Over the next few chapters, we will focus on building an image sharing platform. Users will be able to bookmark any image on the Internet and share it with others. They will also be able to see activity on the platform from the users they follow and like/unlike the images shared by them.

In this chapter, we will start by creating a functionality for users to log in, log out, edit their password, and reset their password. You will learn how to create a custom profile for your users and you will add social authentication to your site.

This chapter will cover the following topics:

- Using the Django authentication framework
- Creating user registration views
- Extending the user model with a custom profile model
- Adding social authentication with Python Social Auth

Let's start by creating your new project.

Creating a social website project

You are going to create a social application that will allow users to share images that they find on the Internet. You will need to build the following elements for this project:

- An authentication system for users to register, log in, edit their profile, and change or reset their password
- A follow system to allow users to follow each other on the website
- A functionality to display shared images and implement a bookmarklet for users to share images from any website
- An activity stream that allows users to see the content uploaded by the people that they follow

This chapter will address the first point on the list.

Starting your social website project

Open the terminal and use the following commands to create a virtual environment for your project and activate it:

```
mkdir env
python3 -m venv env/bookmarks
source env/bookmarks/bin/activate
```

The shell prompt will display your active virtual environment, as follows:

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

Install Django in your virtual environment with the following command:

```
pip install Django==3.0.*
```

Run the following command to create a new project:

```
django-admin startproject bookmarks
```

The initial project structure has been created. Use the following commands to get into your project directory and create a new application named `account`:

```
cd bookmarks/
django-admin startapp account
```

Remember that you should add the new application to your project by adding the application's name to the `INSTALLED_APPS` setting in the `settings.py` file. Place it in the `INSTALLED_APPS` list before any of the other installed apps:

```
INSTALLED_APPS = [  
    'account.apps.AccountConfig',  
    # ...  
]
```

You will define Django authentication templates later on. By placing your application first in the `INSTALLED_APPS` setting, you ensure that your authentication templates will be used by default instead of any other authentication templates contained in other applications. Django looks for templates by order of application appearance in the `INSTALLED_APPS` setting.

Run the next command to sync the database with the models of the default applications included in the `INSTALLED_APPS` setting:

```
python manage.py migrate
```

You will see that all initial Django database migrations get applied. Next, you will build an authentication system into your project using the Django authentication framework.

Using the Django authentication framework

Django comes with a built-in authentication framework that can handle user authentication, sessions, permissions, and user groups. The authentication system includes views for common user actions such as log in, log out, password change, and password reset.

The authentication framework is located at `django.contrib.auth` and is used by other Django `contrib` packages. Remember that you already used the authentication framework in *Chapter 1, Building a Blog Application*, to create a superuser for your blog application to access the administration site.

When you create a new Django project using the `startproject` command, the authentication framework is included in the default settings of your project. It consists of the `django.contrib.auth` application and the following two middleware classes found in the `MIDDLEWARE` setting of your project:

- `AuthenticationMiddleware`: Associates users with requests using sessions
- `SessionMiddleware`: Handles the current session across requests