

CH3: Extending Your Blog Application

Functional overview:



We will build the functionality to add tags to posts. We will extend the `post_list` view to filter posts by tag. When loading a single post in the `post_detail` view, we will retrieve similar posts based on common tags. We will also create custom template tags to display a sidebar with the total number of posts, the latest posts published, and the most commented posts.

We will add support to write posts with Markdown syntax and convert the content to HTML. We will create a sitemap for the blog with the `PostSitemap` class and implement an RSS feed with the `LatestPostsFeed` class.

Finally, we will implement a search engine with the `post_search` view and use PostgreSQL full-text search capabilities.

The source code for this chapter can be found at
<https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter03>.

You can install all the requirements at once with the command **python -m pip install r - .requirements.txt**

Implementing tagging with django-taggit

A very common functionality in blogs is categorizing 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 need to install django-taggit via pip by running the following command

```
python -m pip install django-taggit==5.0.1
```

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

```
INSTALLED_APPS = [  
    #####  
, 'taggit'  
, 'blog.apps.BlogConfig'  
]
```

Note:

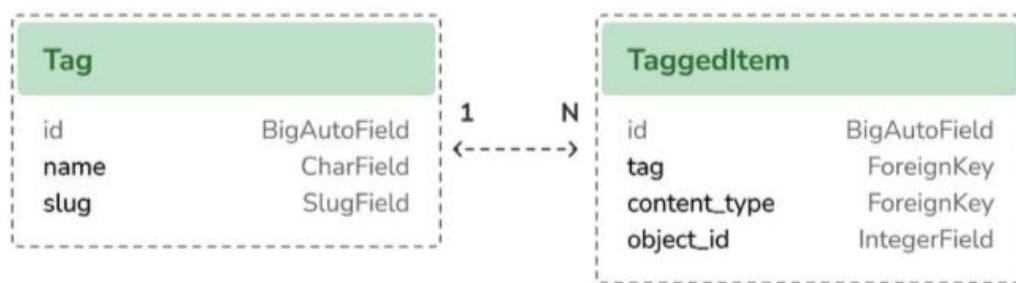
- ★ It's good practice to keep the Django packages at the top, third-party packages in the middle, and local applications at the end of INSTALLED_APPS.

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:



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.

دعيني أشرح لك مفهوم Tag و TaggedItem بطريقة مبسطة مع مثال عملي بناءً على الرسم البياني اللي وفرتنيه والكود اللي كنت تعملني عليه.

شرح بسيط لـ Tag و TaggedItem

- **Tag** (tags) هو جدول في قاعدة البيانات بيخزن الأوصمة: (الوسم) اللي بيستخدمها لتصنيف المحتوى. كل وسم بيكون عنده (tags) رقم تعريفه فريد: **id**.
اسم الوسم (مثل "رياضة" أو "طبخ") : **name**.
مثلاً "رياضة" بتصير (name) نسخة مختصرة ومناسبة للروابط من الـ: **slug** "sports".
 - يعني لو عندك مقال عن كرة القدم، ممكن تضيفيه وسم "كرة_قدم".
 - **TaggedItem** (العنصر الموسوم) هو جدول بيربط بين الوسم (Tag) وبين الكائن اللي بتجيبي تضيفيه الوسم (مثل مقال أو Tag) وبين المدونة (العنصر الموسوم).
يعني لو عندك مقال عن كرة القدم، ممكن تضيفيه وسم "كرة_ القدم".
يعني بيحدد أي وسم بيتم استخدامه ، **tag** بيشير لجدول (ForeignKey) مفتاح أجنبي : **content_type** .
من موديل المدونة بتاعتك؟ **Post** مثلاً، هل هو بيحدد نوع الكائن اللي بتسميته : **object_id** .
اللي هو **Post** مثلاً، رقم (id) رقم تعرف الكائن اللي بتسميته .

يعني **TaggedItem** بيشتغل كجسر بيربط بين الوسم وبين أي كائن (زي المنشورات بقاعدتك).

- العلاقة بينهم:

- العلاقة هي "واحد لكثير" (N:1): وسم واحد (**Tag**) ممكن يرتبط بكل عنصر موسوم (**TaggedItem**).
 - يعني الوسم "كرة_قدم" ممكن يرتبط بمنشور رقم 1 ومنشور رقم 3 ومنشور رقم 10، وكل علاقه من دول بتحزن كسجل في جدول **TaggedItem**.

مثال تبسيطى:

افتراضي عندك مدونة زي اللي بتعمل علىها، وفيها منشورات (**Post**). عندك منشورين:

- منشور 1: عنوانه "مباراة اليوم"، وبتحبى تضيفيله أوسمة "كرة_قدم" و"رياضة".
- منشور 2: عنوانه "وصفة كيك"، وبتحبى تضيفيله وسم "طبخ".

1. جدول **Tag**

بيتم إنشاء سجلات للأوسمة:

2. جدول **TaggedItem**

بيتم إنشاء سجلات للأوسمة:

id	name	slug
1	كرة_قدم	football
2	رياضة	sports
3	طبخ	cooking

3. جدول **TaggedItem**

بيتم ربط كل منشور بالأوسمة الخاصة فيه:

id	tag يشير لـ Tag.id	content_type (نوع الكائن)	object_id (رقم الكائن)
1	1 (كرة_قدم)	Post	1 (منشور "مباراة اليوم")
2	2 (رياضة)	Post	1 (منشور "مباراة اليوم")
3	3 (طبخ)	Post	2 (منشور "وصفة كيك")

المثال:

شرح المثال:

- المنشور "مباراة اليوم" (رقمه 1) مربوط بوسمين: "كرة_قدم" و"رياضة". فبيكون عنده سجلين في جدول `.TaggedItem`.
- المنشور "صفة كيك" (رقمه 2) مربوط بوسم واحد: "طبخ". فبيكون عنده سجل واحد في جدول `.TaggedItem`.

مين بيكون `? TaggedItem`

يعني. (في حالتك `Post` زي الـ) وبين الكائن اللي بتسميه (`Tag`) هو ببساطة السجل اللي بيربط بين الوسم `TaggedItem`

- الكائن الموسوم (زي الـ `Post`) مش هو `TaggedItem` نفسه، لكن `TaggedItem` هو اللي بيخليك تربطني بالوسم `Post`.
- في الكود بتاعك، لما استخدمنت `TaggableManager` في موديل `Post` في موديل `TaggableManager` لما استخدمنت `TaggableManager` في موديل `Post`.

python

 Copy

```
tags = TaggableManager()
```

ده بيخلி `django-taggit` ينشئ العلاقات دي تلقائياً في جدول `TaggedItem` لما تضيفي أوسمة لمنشور.

مسن حسيبي سي اندور.

لو عندك منشور في مدونتك:

python

 Copy

```
post = Post.objects.create(title="مباراة اليوم", slug="match-today", body="تفاصيل المباراة")
post.tags.add("كرة_قدم", "رياضة")
```

هنا `django-taggit` هيضيف سجلين في جدول `TaggedItem` عشان يربط المنصور ده بالوسمين "كرة_قدم" و"رياضة".

ليش بنستخدمهم؟

- عشان نقدر نصنف المحتوى بسهولة. مثلاً، لو عايزه تجمعي كل المنشورات اللي فيها وسم "رياضة":

python

    Copy

```
posts = Post.objects.filter(tags__name__in=["رياضة"])
```

بيساعد في تنظيم المحتوى وتحسين البحث والتصفيه في المدونة.

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

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

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
```

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:

	NAME	SLUG
<input checked="" type="checkbox"/>	django	django
<input type="checkbox"/>	jazz	jazz
<input type="checkbox"/>	music	music

3 tags

Figure 3.3: The tag change list view on the Django administration site

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

Content type:	Blog post
Object ID:	1

Figure 3.4: The tag edit view on the Django administration site

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:

Tags:
A comma-separated list of tags.

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">
```

The join template filter works analogously to Python's string join() method. You can concatenate a list of items into one string, using a specific character or string to separate each item. For example, a list of tags like ['music', 'jazz', 'piano'] is converted into a single string, 'music, jazz, piano', by joining them with ',' as the join() separator.

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:

Who was Django Reinhardt?

Tags: [music](#) , [jazz](#)

Published Jan. 1, 2024, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was one of the first major jazz talents to emerge from ...

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)  
    #####  
    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.

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 `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//', 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:

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

1

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 %}
```

```
My Blog
```

```
{% if tag %}
```

```
    Posts tagged with "{{ tag.name }}"
```

```
{% endif %}
```

```
{% for post in posts %}
```

```
#####
```

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.

```
#####
```

```
<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 %}
```

```
#####
```

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 with 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:

Posts tagged with "jazz"

Who was Django Reinhardt?

Tags: [music](#) , [jazz](#)

Published Jan. 1, 2024, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was one of the first major jazz talents to emerge from ...

Page 1 of 1.

Retrieving posts by similarity

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. Let's edit the `post_detail` view to incorporate these similarity-based post suggestions.

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

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 tuple 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. You pass the `similar_posts` object to the context dictionary for the `render()` function.

الهدف العام من الكود

الهدف هو إيجاد منشورات مشابهة للمنشور الحالي بناءً على الأوسمة المشتركة. يعني لو عندك منشور عن "جاز" وموسم بـ "music" و "jazz". الكود هيحاول يلاقي منشورات تانية عندها نفس الأوسمة، ويرتبها بناءً على عدد الأوسمة المشتركة و تاريخ النشر، ويجيب أول 4 منشورات فقط.

الخطوات بالتفصيل

1. استخراج معرفات الأوسمة للمنشور الحالي

```
python
post_tags_ids = post.tags.values_list('id', flat=True)
```

ما اللي بيحصل هنا؟ بنسخرج قائمة بمعرفات (IDs) الأوسمة اللي متعلقة بالمنشور الحالي (post).
الدوال المستخدمة:

- `post.tags` :
- دي مش دالة، دي حقل في موديل `Post` بتاعك، وهو من نوع `TaggableManager` (من مكتبة `Taggit`).
- وظيفته: بربط المنشئ، بالأوسمة المرتبطة به فـ، حدوا، `TaggedItem`. بعند، له المنشئ، عنده أوسمة؛،

- وظيفته: يربط المنشور بالأوسمة المرتبطة بيها في جدول **TaggedItem**. يعني لو المنشور عنده أوسمة زي "jazz" و "music" بيسمح لك تتعامل معها بسهولة. هنا بيتم استخدامه عشان نجيب الأوسمة الخاصة بالمنشور.

```
.values_list('id', flat=True) :
```

- **الباراميترات:** `get_queryset()` `get_object_or_404()`
 - **وظيفتها:** بتجيب قائمة من القيم لـ `QuerySet` `di`

- اسما الحقل اللي عايزين نجيب قيمة (معرف الوسم) : `'id'`
 - بدل ما تكون قيمة من (flat list) بيخلى النتيجة قائمة بسيطة : `flat=True` tuples.

مثال:

- من غير `flat=True`، لو عندك أوسمة IDs باتباعتها `[1, 2, 3]`، النتيجة هتبقي `[(), 2, (), 1]`.

- مع `flat=True` ، النتيجة هتبقي `[1, 2, 3]` (أبسط وأسهل في الاستخدام).

- فائدتها هنا: بنحتاج قائمة بسيطة من IDs عشان نستخدمها في الخطوة الجاية للبحث عن منشورات تانية عندها نفس الأوصمة.

النتيجة: لو المنشور الحالي عنده أوصمة بـ [1, 2, 3] IDs (مثلاً "music" و "jazz")، المتغير `post_tags_ids` هيقي:

النتيجة: لو المنشور الحالي عنده أوسمة بـ `post_tags_ids` هيبقى: [1, 2] مثلاً "jazz" و "music" IDs (["jazz", "music"])

2. البحث عن المنشورات اللي عندها أوسمة مشتركة

```
python Copy  
  
similar_posts = Post.published.filter(  
    tags__in=post_tags_ids  
)  
.exclude(id=post.id)
```

- ما اللي بيحصل هنا؟ بنبحث عن كل المنشورات المنشورة (published) اللي عندها أي وسم من الأوسمة بتاعت
المنشور الحالي، وبنستثني المنشور الحالي نفسه.
 - **الدوال المستخدمة:**

Post.published :

• ده مش دالة، ده Manager مخصص في موديل Post بتعاك، وانت عرفتیه في الكود:

• ده مشن دالة، ده `Post` مخصوص في موديل `Manager` بتعالك، وانت عرفتنيه في الكود:

python

Copy

```
published = PublishedManager()
```

• وظيفتها: بيرجع `QuerySet` لكل المنشورات اللي حالتها `PUBLISHED` فقط (يعني مش هيجبب المنشورات اللي حالتها `DRAFT`).

• فائدتها: بيضمن إن المنشورات اللي بنجيبيها تكون منشورة فعلاً ومتحركة للعرض.

• `.filter(tags__in=post_tags_ids)`:

• دي دالة من `Django QuerySet`.

• وظيفتها: بتتصفي النتائج بناءً على شرط معين.

• الباراميتر `tags__in`:

• `tags` اللي هو `Post` ده اسم الحقل في موديل `TaggableManager`.

• `__in` (lookup operator) يعني بنقول لـ `post_tags_ids` عندة أوسمة تكون موجودة في القائمة.

• أو أي قائمة موجودة في (يعني هنا بنقول: جيب المنشورات اللي عندها أي وسم من `[1, 2, post_tags_ids]`).

• فائدتها: بتساعدنا نلاقي المنشورات اللي عندها أوسمة مشتركة مع المنشور الحالي.

• فائدتها: بتساعدنا نلاقي المنشورات اللي عندها أوسمة مشتركة مع المنشور الحالي.

• `.exclude(id=post.id)`:

• دي دالة من `Django QuerySet`.

• وظيفتها: بتسئلي السجلات اللي بتطابق الشرط.

• الباراميتر `id=post.id` : بتقول استثنى المنشور اللي معرفه (`id`) يساوي معرف المنشور الحالي.

• فائدتها: بنسخدمها عشان ما نجيبيش المنشور الحالي نفسه في قائمة المنشورات المشابهة.

• النتيجة: لو في منشورات تانية عندها أوسمة زي "jazz" أو "music" (يعني IDs بتعايتها 1 أو 2)، هتتضاف للـ `similar_posts`. بس لو المنشور الحالي نفسه (اللي ID بتعاه مثلًا 5) كان موجود في النتائج، هيتحذف من القائمة `.exclude` بسبب.

3. إضافة حقل محسوب لعدد الأوسمة المشتركة

python

Copy

```
similar_posts = similar_posts.annotate(
    same_tags=Count('tags')
)
```

ما اللي بيحصل هنا؟ بنضيف حقل جديد (مش موجود في الموديل) اسمه `same_tags` لكل منشور في الـ

3. إضافة حقل محسوب لعدد الأوسمة المشتركة

python

Copy

```
similar_posts = similar_posts.annotate(  
    same_tags=Count('tags')  
)
```

- ما اللي بيحصل هنا؟ بنضيف حقل جديد (مش موجود في الموديل) اسمه `same_tags` لكل منشور في الـ `similar_posts`. الحقل ده هيحسب عدد الأوسمة المشتركة بين المنشور ده والأوسمة اللي اخترناها في `.post_tags_ids` . الدوال المستخدمة:
 - `.annotate(same_tags=Count('tags'))` :
 - دي دالة من `Django QuerySet`
 - وظيفتها: بتضيف حقل محسوب (calculated field) لكل سجل في الـ `QuerySet` (aggregation) لـ `Count('tags')` من `Django`.
 - الباراميترات:
 - اسم الحقل الجديد اللي هنضيفه: `same_tags`
 - `Count('tags')` دي دالة تجمع (aggregation) دالة `Count`.

4. ترتيب المنشورات و اختيار أول 4 فقط

python

 Copy

```
similar_posts = similar_posts.order_by('-same_tags', '-publish')[4:]
```

- ما اللي بيحصل هنا؟ بترتيب المنشورات المشابهة بناءً على عدد الأوسمة المشتركة (من الأكبر للأصغر)، ولو في منشورات عندهم نفس عدد الأوسمة المشتركة، بترتيب المنشورات دي بناءً على تاريخ النشر (من الأحدث للأقدم). وبعدين بنأخذ أول 4 منشورات فقط.
 - الدوال المستخدمة:

```
.order_by('-same_tags', '-publish')
```
 - دي دالة من Django QuerySet .
• وظيفتها: بترتيب النتائج بناءً على حقول معينة.
 - الباراميترات:
 - بشكل تنازلي (الـ `annotate` اللي ضفناه في الـ `same_tags` بترتيب بناءً على الحقل : `'-same_tags'` الـ `descending`).
 - يعني تنازلي (الـ `-`).
 - بترتيب بناءً على حقل ، `'-publish'` لو في منشورات عندهم نفس قيمة : `'same_tags'` .
• بشكل تنازلي (يعني الأحدث أولاً).

□

• دالة Count •

- وظيفتها: بتحسب عدد العناصر في حقل معين.
 - الباراميتر **'tags'** : بتقول احسب عدد الأوسمة المرتبطة بكل منشور في الـ `similar_posts` اللي عملناها قبل كده، الـ ملاحظة: بسبب الـ `filter(tags_in=post_tags_ids)` هنا هيحسب بس عدد الأوسمة اللي موجودة في `post_tags_ids` (مش كل `Count('tags')` الأوسمة بتاعت المنشور).
 - فائدتها: بتساعدنا نعرف عدد الأوسمة المشتركة بين كل منشور مشابه والمنشور الحالي، وده هيبقى مهم عشان نرتب المنشورات بعدين.
 - مثال:

لو المنشور الحالي عنده أوسمة بـ `[1, 2]` (يعني "music" و "jazz"). وفي منشور مشابه (A) عنده أوسمة `[1, 3]` (يعني "music" و "rock"). فال `same_tags` هتبقى `1` (لأن "music" وسم واحد مشترك). ومنشور مشابه (B) عنده أوسمة `[1, 2]` (يعني "jazz" و "music"). فال `same_tags` هتبقى `2` (لأن في وسمين مشتركين).

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

Similar posts

{% for post in similar_posts %}

```
<p>  
    <a href="{{ post.get_absolute_url }}>{{ post.title }}</a>  
</p>  
{% empty %}  
  
    There are no similar posts yet.  
  
{% endfor %}
```

Who was Django Reinhardt?

Published Jan. 1, 2024, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was one of the first major jazz talents to emerge from Europe and remains the most significant.

[Share this post](#)

Similar posts

There are no similar posts yet.

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:

Change post

Who was Miles Davis?

[HISTORY](#)[VIEW ON SITE >](#)

Title: Who was Miles Davis?

Slug: who-was-miles-davis

Author: 1 Q admin

Body: Miles Davis was an American jazz musician, trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.

Publish: **Date:** 2024-01-02 Today |

Time: 13:00:00 Now |

Note: You are 2 hours ahead of server time.

Status: Published

Tags: jazz, music

A comma-separated list of tags.

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

Change post

Notes on Duke Ellington

[HISTORY](#)[VIEW ON SITE >](#)

Title: Notes on Duke Ellington

Slug: notes-on-duke-ellington

Author: 1 Q admin

Body: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century.

Publish: **Date:** 2024-01-02 Today |

Time: 16:00:00 Now |

Note: You are 2 hours ahead of server time.

Status: Published

Tags: jazz

A comma-separated list of tags.

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

Who was Django Reinhardt?

Published Jan. 1, 2024, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was one of the first major jazz talents to emerge from Europe and remains the most significant.

[Share this post](#)

Similar posts

[Who was Miles Davis?](#)

[Notes on Duke Ellington](#)

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 readers. django-taggit also includes a `similar_objects()` manager that you can use to retrieve objects by shared tags.

Creating custom template tags and filters

Django offers a variety of built-in template tags, such as `{% if %}` or `{% block %}`.

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 a list of the 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, which 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 because 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 on 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.

6

1. ما هو register وما فائدته؟

- وهو `template.Library` يتم تعريفه كمثل للكلاس `Django`. هو متغير يتم إنشاؤه كجزء من مكتبة القوالب في `django.template` (المدمجة `Django`)، بدون `templatetags` المخصصة التي تُنشئها في ملف `(tags)` والفلاتر `(filters)` للتسجيل الوسوم `register` الفاندة يستخدم على الوسوم أو الفلاتر التي تحاول إنشاءها `Django` هذا المتغير، لن يتعرف وبين نظام القوالب في `blog_tags.py` هو الجسر الذي يربط بين الدول التي تكتبها في ملف `register`، بمعنى آخر `Django` بحيث يمكن استخدام هذه الدول كوسوم أو فلاتر داخل قوالب `HTML`.

مثال:

python

... Copy

```
from django import template
register = template.Library() # إنشاء مثيل من template.Library
```

هنا، يتم إنشاء register ليصبح الكائن الذي يتم من خلاله تسجيل الوسوم.

2

2. ما هو الديكوريتور register.simple_tag@ وما فائدته؟

- الديكوريتور `register.simple_tag@` هو أداة تُستخدم لتسجيل دالة Python عاديّة كوسّم بسيط في نظام قوالب Django.
 - **الفائدة:**
 - يأخذ الدالة التي تُعرفها (مثلاً `total_posts`) ويحولها إلى وسم يمكن استخدامه في قوالب Django.
 - يخبر Django أن هذه الدالة يجب أن تُعامل كوسّم قالب، ويستخدم اسم الدالة كاسم الوسم (في هذه الحالة، `total_posts`).
 - يسمح لك بتمرير قيمة الإرجاع (return value) من الدالة إلى القالب مباشرة.
 - **كيف يعمل؟:**
 - عندما تُضع `register.simple_tag@` فوق دالة، فإن Django يسجل هذه الدالة في مكتبة القوالب `register`، وتُصبح متاحة لل استخدام في قوالب HTML.
 - على سبيل المثال، في الكود:

python

... Copy

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



total_posts `{% total_posts %}` وسماً يمكن استدعاءه في القالب باستخدام `total_posts` .

3. ما هو الـ `simple_tag` ؟

- الوسم البسيط (`simple tag`) هو نوع من الوسوم في Django يُستخدم لإجراء عمليات بسيطة وإرجاع قيمة مباشرة إلى القالب.
- خصائصه:
 - يأخذ الدالة التي تحته ويرجع قيمتها كنص (`string`) أو أي نوع بيانات آخر يمكن عرضه في القالب.
 - يمكن أن يأخذ وسائط (`arguments`) إذا تم تعريفها في الدالة.
 - لا يسمح بمعالجة معقدة لمحظى القالب (مثل تحليل كتلة من النصوص داخل الوسم)، وهذا يختلف عن أنواع أخرى من الوسوم مثل `block tag` أو `inclusion_tag` .
- مثال:

- الدالة `total_posts` تحسب عدد المنشورات المنشورة (`Post.published.count`) وترجع هذا العدد.
- عند استدعاء `{% total_posts %}` في القالب، سيتم عرض العدد مباشرة (مثلاً `42` إذا كان هناك 42 منشوراً).

كيف يظهر الوسم في القالب؟

4. كيف يظهر الوسم في القالب؟

- لاستخدام الوسم في قالب HTML، تحتاج إلى تحميل مكتبة الوسوم أولاً باستخدام `load` ، ثم استدعاء الوسم باسمه.
- خطوات الاستخدام:
 - في ملف القالب (مثلاً `base.html` أو أي قالب آخر)، قم بتحميل مكتبة الوسوم:

html

...



`{% load blog_tags %}`



بدون الامتداد `blog_tags` هو اسم ملف `blog_tags.py`.

- استخدم الوسم في المكان الذي تريده عرض القيمة فيه:

html

...



`<p>Total posts: {% total_posts %}</p>`

- عند تحميل الصفحة، سيتم استبدال `{% total_posts %}` بالقيمة التي تُرجعها الدالة (مثلاً `42`).

النتيجة:

5. لماذا نضع `register.simple_tag` في الديكوريتور؟

• السبب:

- الديكوريتور هو طريقة Python لتعديل سلوك الدالة أو إضافة وظائف إليها.
- في هذه الحالة، يُخبر Django أن الدالة التي تحته يجب أن تُسجل كوسم قالب، ويربطها بمكتبة القوالب `(register.register.simple_tag)`.
- بدون الديكوريتور، ستكون الدالة مجرد دالة Python عادية ولن يتعرف عليها Django كوسم يمكن استخدامه في القوالب.

• بديل للديكوريتور:

- يمكن تسجيل الدالة يدوياً بدون ديكوريتور باستخدام:

```
python
...
register.simple_tag(total_posts)
```

لكن استخدام الديكوريتور أكثر شيوعاً وأنظف من حيث الكود.

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')`.

Note:

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 our 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 %}

####

<p>

This is my blog.

I've written {% total_posts %} posts so far.
```

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 3.12: 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:

```
TemplateSyntaxError at /blog/
'blog_tags' is not a registered tag library. Must be one of:
admin_list
admin_modify
admin_urls
blog_tags
cache
i18n
l10n
log
static
tz
```

..

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`.

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

In the preceding code, you have added 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

```
#####
This is my blog.
I've written {% total_posts %} posts so far.
</p>
<h3> Latest posts </h3>
{% show_latest_posts 3 %}
```

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:

My blog

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

Latest posts

- [Notes on Duke Ellington](#)
- [Who was Miles Davis?](#)
- [Who was Django Reinhardt?](#)

1. ما هو الـ `inclusion_tag` ؟

- **الوسم التضميني (inclusion tag)** هو نوع من الوسوم المخصصة في Django يُستخدم لتقديم قالب (template) مع سياق (context) مخصوص. بدلاً من إرجاع قيمة بسيطة (مثل الـ `simple_tag`)، يقوم الـ `inclusion_tag` بإرجاع قاموس (dictionary) يحتوي على بيانات تُستخدم لعرض قالب معين.
- **الفكرة الأساسية:** بدلاً من كتابة HTML مباشرة في القالب الرئيسي أو تكرار كود HTML في عدة أماكن، يمكنك إنشاء وسم يقوم بإنشاء جزء من واجهة المستخدم (مثل قائمة منشورات) وعرضه في أي مكان تريده في القوالب.

2. فائدة الـ `inclusion_tag`

- **إعادة الاستخدام:** يسمح بإعادة استخدام جزء من واجهة المستخدم (مثل قائمة المنشورات الأخيرة) في قوالب متعددة دون تكرار الكود.
- **المرونة:** يمكنك تمرير بيانات ديناميكية (مثل عدد المنشورات أو قائمة المنشورات) إلى القالب المرتبط بالوسم.
- **التنظيم:** يفصل منطق جلب البيانات (مثل استعلام قاعدة البيانات) عن العرض (HTML)، مما يجعل الكود أكثر تنظيماً وصيانة.
- **التخصيص:** يمكنك تخصيص القالب المرتبط بالوسم (مثل `latest_posts.html`) لتحديد شكل العرض، بينما يتحكم الوسم في البيانات التي يتم تمريرها.

```
python
@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}
```

- `@register.inclusion_tag('blog/post/latest_posts.html')`:
 - هذا الديكوريتور يسجل الدالة `show_latest_posts` كـ `.inclusion_tag` في `blog/post/latest_posts.html` عند استدعاء الوسم.
- `def show_latest_posts(count=5)`:
 - الدالة تأخذ وسيطًا اختياريًا `count` (القيمة الافتراضية هي 5)، والذي يحدد عدد المنشورات التي سيتم جلبها.
- `latest_posts = Post.published.order_by('-publish')[:count]`:
 - يتم جلب المنشورات المنشورة (`Post.published`) مرتبة حسب تاريخ النشر بترتيب تنازلي (`-publish`)، ويتم أخذ أول `count` منشورات فقط.
- `return {'latest_posts': latest_posts}`:
 - الدالة تُرجع قاموسًا يحتوي على مفتاح `latest_posts` وقيمه هي قائمة المنشورات التي تم جلبها.

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):
    """Returns the most commented posts."""

    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.

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

```
#####  
{% 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>
```

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.

شرح الفقرة

الفقرة تتحدث عن إنشاء تاج قالب (Template Tag) في إطار عمل Django، وهو تاج بسيط يُستخدم لاسترجاع قائمة من المنشورات (Posts) التي تحتوي على أكبر عدد من التعليقات في مدونة، وتخزين هذه القائمة في متغير يمكن استخدامه لاحقاً في القالب (Template). الهدف هو عرض المنشورات الأكثر تعليقاً في واجهة المستخدم.

1. الكود في ملف blog_tags.py

```
python ... ⌂ Copy

from django.db.models import Count

@register.simple_tag
def get_most_commented_posts(count=5):
    """Returns the most commented posts."""
    return Post.published.annotate(total_comments=Count('comments')).order_by('-total_co
```


• التاج:

- `@register.simple_tag` يُستخدم لتسجيل الدالة كتابع قالب بسيط يمكن استخدامه في قالب Django.
 - `def get_most_commented_posts(count=5)` الدالة تأخذ وسيطًا اختيارياً `count` (القيمة الافتراضية 5) لتحديد عدد المنشورات التي سيتم استرجاعها.

• المنطق:

- يُرجح على مدير مخصص **Post** يفترض أن نموذج **published** يسمى (Manager) يحتوي على مديري المنشورات فقط (مثلاً، تلك التي تكون حالتها "منشورة").

- نحسب عدد التعلیقات المرتبطة بكل منشور (يفترض أن هناك علاقة ForeignKey Count('comments')) . Post

۱۰۰

- يحدد عدد المنشورات التي سيتم إرجاعها بناءً على قيمة `[:count]` .

2000

- لكل منشور.

```
① ... ⌂ Copy
{%
    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>
```

• استخدام التاج:

- `{% get_most_commented_posts as most_commented_posts %}`:
- يقوم بتشغيل تاج القالب `.get_most_commented_posts` الذي يحتوي على المنشورات (الـ `QuerySet`) في متغير يُسمى `.most_commented_posts`.
- يخزن النتيجة (الـ `QuerySet`) في متغير يُسمى `.most_commented_posts`.
- استخدام `as` يسمح بتخزين النتيجة بدلاً من عرضها مباشرةً، مما يتتيح إعادة استخدام النتيجة في القالب.

• عرض النتائج:

- يتم استخدام حلقة `for` للتكرار على `most_commented_posts` (الـ `QuerySet` المخزن).
- لكل منشور:

• عرض النتائج:

- يتم استخدام حلقة `for` للتكرار على `most_commented_posts` (الـ `QuerySet` المخزن).
- لكل منشور:

• يتم عرض عنوان المنشور (`post.title`) داخل رابط (`<a>`) يؤدي إلى عنوان URL الخاص بالمنشور (`post.get_absolute_url`).

• يفترض أن نموذج `Post` يحتوي على دالة `get_absolute_url` تُرجع الرابط الدائم (permalink) للمنشور.

• المنشورات تُعرض في قائمة غير مرتبة (``).

• ملاحظة إضافية:

- السطر `{% show_latest_posts 3 %}` يُشير إلى تاج قالب آخر (غير موضح في الفقرة) يُستخدم لعرض أحدث 3 منشورات. هذا التاج ليس له علاقة مباشرةً بـ `get_most_commented_posts` ولكنه جزء من القالب.

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 `djang0`, 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.

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.

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

```
python -m pip install markdown==3.6
```

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

```
import markdown

from django.utils.safestring import mark_safe

@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, is converted to <p> (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.

Note:

In Django, HTML content is escaped by default for security. Use `mark_safe` cautiously, only on content you control. Avoid using `mark_safe` on any content submitted by nonstaff users to prevent security vulnerabilities.

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 %}

#####
{{ post.body|markdown }}

#####
```

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.

Note:

Storing text in Markdown format in the database, rather than HTML, is a wise security strategy. Markdown limits the potential for injecting malicious content. This approach ensures that any text formatting is safely converted to HTML only at the point of rendering the template.

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 %}

#####
{{ 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:

Add post

Title:	Markdown post
Slug:	markdown-post
Author:	1
Body:	<p>This is a post formatted with markdown</p> <p>*This is emphasized* and **this is more emphasized**.</p> <p>Here is a list:</p> <ul style="list-style-type: none">* One* Two* Three <p>And a [link to the Django website](https://www.djangoproject.com/).</p>
Publish:	Date: 2024-01-02 Today <input type="button" value="Calendar"/>
	Time: 16:30:00 Now <input type="button" value="Clock"/>
<small>Note: You are 2 hours ahead of server time.</small>	
Status:	Published <input type="button" value="Select"/>
Tags:	markdown
<small>A comma-separated list of tags.</small>	

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 `sitemap` applications in your project. We are going to build a sitemap for the blog that includes the links to all published posts.

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
```

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

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.

We have created the sitemap. Now we just need to create a 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.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 defined a `sitemaps` dictionary. Multiple sitemaps can be defined for the site. We have defined a URL pattern that matches the `sitemap.xml` pattern and uses the `sitemap` view provided by Django. The `sitemaps` dictionary is passed to the `sitemap` view.

Start the development server 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:

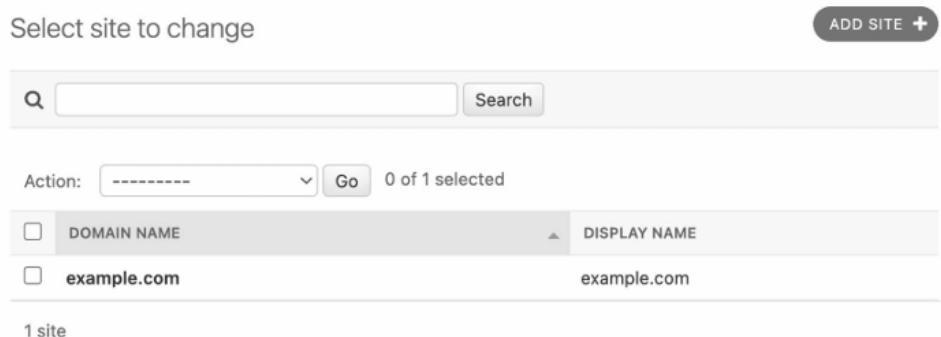
This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <SCRIPT id="allow-copy_script"/>
  <url>
    <loc>http://example.com/blog/2025/1/23/4</loc>
    <lastmod>2025-01-23</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2025/1/23/3</loc>
    <lastmod>2025-04-16</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2025/1/23/2</loc>
    <lastmod>2025-04-16</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2025/1/23/1</loc>
    <lastmod>2025-01-23</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2025/1/23/who-was-django-reinhardt</loc>
    <lastmod>2025-04-19</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

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 data base.

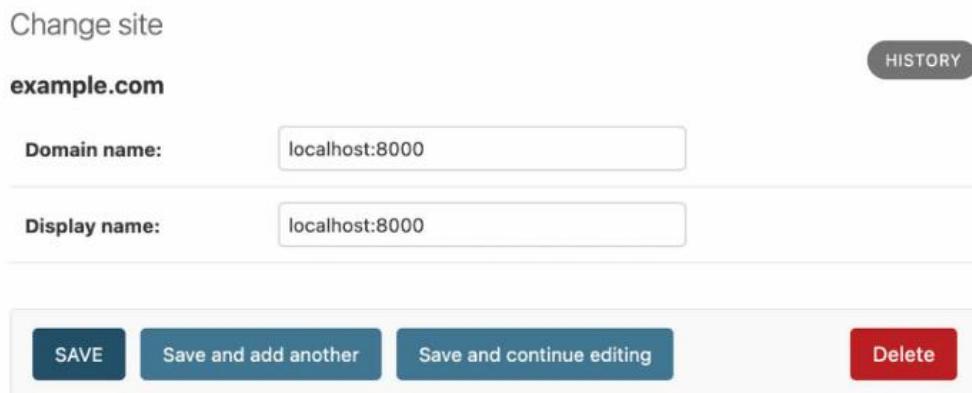
Open <http://127.0.0.1:8000/admin/sites/site/> in your browser. You should see something like this:



Action:	DOMAIN NAME	DISPLAY NAME
<input checked="" type="checkbox"/>	example.com	example.com

1 site

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



Open `http://127.0.0.1:8000/sitemap.xml` in your browser again. The URLs displayed in your sitemap will now use the new hostname and look like `http://localhost:8000/blog/2024/1/22/ 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.

شرح عام عن الموضوع (Django في Sitemap) ما هو Sitemap؟

الـ Sitemap هو ملف XML يُستخدم لإخبار محركات البحث (مثل Google) عن الصفحات الموجودة في موقعك الإلكتروني، ومدى أهميتها، وكم مرة يتم تحميلها. الهدف منه هو مساعدة محركات البحث على فهرسة (indexing) محتوى موقعك بشكل أفضل، مما يزيد من احتمال ظهور موقعك في نتائج البحث. بمعنى آخر، الـ Sitemap يشبه "خريطة" لموقعك توجه محركات البحث إلى الصفحات التي تريد منهم زيارتها.

لماذا نستخدم Django في Sitemap؟

في Django، يوفر إطار عمل السـitemap طريقة ديناميكية لإنشاء هذا الملف تلقائياً بناءً على محتوى موقعك (مثـل المقالات أو المنتجات). بدلاً من كتابة ملف XML يدوياً، يمكنك استخدام Django لـ Sitemap لإنشاء المحتوى وروابط صفحـات موقعك (مثـل منشورات المدونة) مع تفاصـيل إضافـية مثل تاريخ آخر تعـديل وأولـوية الصـفحة.

فائـدة هذا الـدرس:

- تحسين SEO (تهـيئة محركات البحث): يساعد الـ Sitemap محركات البحث على اكتـشاف صفحـات موقعـك بـسهولة، مما يـحسن تـرتـيب موقعـك.
- إـدارة المـحتـوى الـدـينـاميـكي: إذا كنت تـدير مـدونـة أو مـوقـعـاً يـتم تـحـديثـه باـسـتمـرارـ، يـضـمن الـ Sitemap تـحـديثـ الروابـط تـلقـائـياً.

تجـربـة مستـخدمـ أـفـضل: عندـما تـظـهـر صـفحـاتـك في محـركـاتـ البحثـ، يـزـدـاد عـدـدـ الزـوارـ.

إـعدادـ اـحـترـافيـ لـمـوقـعـ: إـضـافـة Sitemap هي مـارـسـة قـيـاسـيةـ في تـطـوـيرـ المـوقـعـ الـاحـترـافيـ.

شرح الفقرات خطوة بخطوة بطريقة بسيطة:

1. إعداد إطار العمل (Framework Setup):

- ماذا نفعل؟

نحتاج إلى تفعيل تطبيقي `django.contrib.sitemaps` و `django.contrib.sites` في مشروع Django.

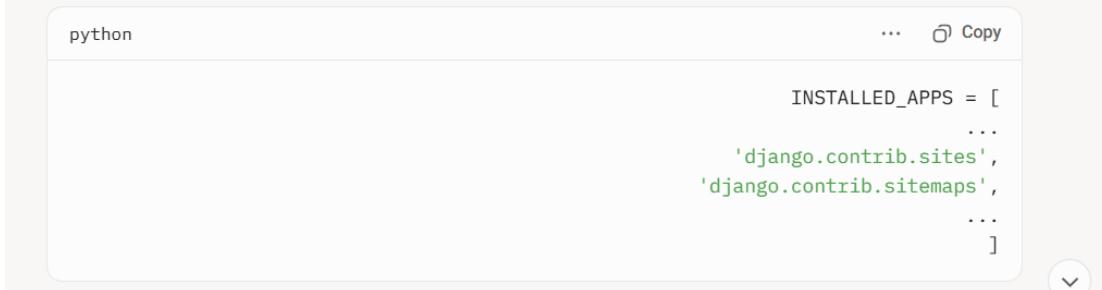
التطبيق الأول (`sites`) يساعد في ربط المحتوى بموقع معين (مفید إذا كنت تدير عدة مواقع في مشروع واحد)،

والثاني (`sitemaps`) هو المسئول عن إنشاء ملف Sitemap.

- كيف نفعل ذلك؟

- افتح ملف `settings.py` في مشروعك.

- أضف التطبيقات إلى قائمة `INSTALLED_APPS`:



```
python
...
Copy
INSTALLED_APPS = [
    ...
    'django.contrib.sites',
    'django.contrib.sitemaps',
    ...
]
```

- أضف `SITE_ID = 1` لتحديد الموقع الافتراضي (كل موقع في Django له رقم تعريفي).

- لماذا نفعل هذا؟

تفعيل هذين التطبيقات يسمح لـ Django بإدارة موقع متعدد وإنشاء ملف Sitemap. بدون `SITE_ID`، لن يعرف

Django أي موقع يرتبط بالـ Sitemap.

- الأمر التالي:

نفذ الأمر `python manage.py migrate sites` لإنشاء الجداول اللازم في قاعدة البيانات لتطبيق `sites`.

2. إنشاء ملف Sitemap:

- ماذا نفعل؟

نشئ ملفًا جديداً باسم `sitemaps.py` داخل تطبيق المدونة (`blog`). هذا الملف سيحتوي على تعريف

لمنشورات المدونة.

2. إنشاء ملف Sitemap

ماذا نفعل؟

ننشئ ملفاً جديداً باسم `sitemaps.py` داخل تطبيق المدونة (`blog`) . هذا الملف سيحتوي على تعريف Sitemap لمنشورات المدونة.

ال코드:

```
python
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
```

شرح الكود:

شرح الكود:

- `from django.contrib.sitemaps import Sitemap`: من `Sitemap` نستورد الكلاس: لإنشاء Django مخصوص `Sitemap`.
- `from .models import Post`: من تطبيق المدونة (`blog`) نستورد نموذج `Post`: لمنشورات.
- `class PostSitemap(Sitemap)`: ننشئ كلاساً جديداً يرث من `Sitemap`.
- `changefreq = 'weekly'`: أو `daily` يمكن أن تكون خبر محركات البحث أن المنشورات تُحدث أسبوعياً: `monthly` (حسب موقعك).
- `priority = 0.9`: نحدد أهمية المنشورات (من 0 إلى 1، حيث 1 هي الأعلى). 0.9 تعني أن المنشورات مهمة جداً.
- `items(self):` هنا نختار جميع المنشورات `Sitemap` هذه الدالة تحدد البيانات التي سيتم تضمينها في `(Post.published.all())` المنشورة.
- `lastmod(self, obj):` من نموذج `updated` نستخدم حقل `updated` (التي يجب أن تكون معرفة في نموذج `Post`) (التي يحدد كل منشور: `Post.get_absolute_url()`) للحصول على رابط كل منشور.

لماذا نفعل هذا؟

هذا الكود يخبر Django كيفية إنشاء Sitemap يحتوي على روابط المنشورات مع معلومات عن تواتر التحديث وأولويتها. يستخدم دالة `Post.get_absolute_url()` (التي يجب أن تكون معرفة في نموذج `Post`) للحصول على رابط كل منشور.

3. إضافة URLs إلى Sitemap

ماذا نفعل؟

نضيف عنوان URL لملف الـ Sitemap (مثلاً `sitemap.xml`) في ملف `urls.py` الرئيسي للمشروع.

ال코드:

python

```
from django.contrib.sitemaps.views import sitemap
from blog.sitemaps import PostSitemap

sitemaps = {'posts': PostSitemap}

urlpatterns = [
    ...
    path('sitemap.xml', sitemap, {'sitemaps': sitemaps}, name='django.contrib.sitemap')
]
```

شرح الكود:

- `from django.contrib.sitemaps.views import sitemap` نستورد عرض: `sitemap` الذي يقدم ملف XML.

شرح الكود:

- `from django.contrib.sitemaps.views import sitemap` الذي يقدم ملف XML.
- `from blog.sitemaps import PostSitemap` الذي أنشأناه.
- `sitemaps = {'posts': PostSitemap}` يمكن إضافة (الخاص بنا Sitemap) ننشئ قاموساً يحتوي على: يمكن إضافة (الخاص بنا Sitemap) ننشئ قاموساً يحتوي على: (الأسماك أخرى مثل المنتجات المزيد إذا كنت تريد).
- `path('sitemap.xml', sitemap, {'sitemaps': sitemaps}, ...)` يحيث عند URL نضيف عنوان: () يتم عرض ملف `/sitemap.xml` زبارة Sitemap.

لماذا نفعل هذا؟

هذا الكود يجعل ملف Sitemap متاحاً عبر رابط مثل `http://127.0.0.1:8000/sitemap.xml`، مما يسمح لمحركات البحث بالوصول إليه.

4. اختبار Sitemap

ماذا نفعل؟

نشغل السيرفر باستخدام `python manage.py runserver`، ثم نزور `http://127.0.0.1:8000/sitemap.xml` في المتصفح.

ماذا سنرى؟

• ماذا سنرى؟

ملف XML يحتوى على روابط جميع المنشورات المنشورة، مع تفاصيل مثل:

- رابط المنشور (مثل [/http://example.com/blog/2024/1/22/markdown-post](http://example.com/blog/2024/1/22/markdown-post)).
- تاريخ آخر تعديل (.lastmod).
- تواتر التحديث (.changefreq).
- الأولوية (.priority).

• لماذا نفعل هذا؟

للتتأكد من أن Sitemap يعمل بشكل صحيح ويحتوى على جميع الروابط التي نريدها.

5. تغيير اسم النطاق (Domain):

• ماذا نفعل؟

بشكل افتراضي، يستخدم Django النطاق `example.com` (لبيئة المحلية):

- افتح لوحة الإدارة ([/http://127.0.0.1:8000/admin](http://127.0.0.1:8000/admin)).
- انتقل إلى قسم `Sites`، وغير النطاق من `example.com` إلى `localhost:8000`.

• لماذا نفعل هذا؟

للتتأكد من أن الروابط في Sitemap تعمل في بيئتك المحلية. في الإنتاج، ستستخدم نطاق موقعك الحقيقي (مثل mysite.com).

لماذا هذا الدرس مهم وما تأثيره؟

- التأثير على SEO: يساعد Sitemap محركات البحث على فهم هيكل موقعك، مما يزيد من فرص ظهور صفحاتك في نتائج البحث.
- التأثير على المستخدمين: كلما تحسنت فهرسة موقعك، زاد عدد الزوار.
- التأثير على التطوير: يعلمك هذا الدرس كيفية استخدام إطار عمل Django لإنشاء ميزات ديناميكية، وهي مهارة أساسية في تطوير المواقع.
- المرونة: يمكنك تخصيص Sitemap ليشمل أنواع محتوى أخرى (مثلا المنتجات أو الصفحات الثابتة) بنفس الطريقة.

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.

ما هو "Feed" (الخلاصة أو النغذية)؟

الـ "Feed" (مثل RSS أو Atom) هو طريقة لعرض آخر التحديثات أو المحتوى الجديد من موقعك (مثل منشورات المدونة) بتنسيق بسيط يمكن للأشخاص الاشتراك فيه. بمعنى آخر، إذا كنت تدير مدونة، يمكن للقراء استخدام برامح خاصة (مثل Feedly أو متصفحات معينة) لتلقي إشعارات تلقائية بكل منشور جديد دون الحاجة لزيارة موقعك يدوياً.

كيف يعمل هذا في Django؟

تساعدك على إنشاء هذه الخلاصات بسهولة، بدلاً من Syndication Feed Framework يحتوي على أداة مدمجة تسمى Django يحتوي على آخر المنشورات في مدونتك. يولد ملف XML يحتوي على المحتوى، أو يمكنك استخدام برنامج مثل Feedly.

خطوات مبسطة لإنشاء Feed في Django

- إنشاء كلاس للـ Feed: تقوم بكتابة كود يحدد ما الذي تريد إظهاره في الخلاصة (مثل عنوان المنشور، الوصف، وتاريخ النشر).
- ربط URL Feed بالـ URL: تضيف رابطاً في موقعك (مثل `yourwebsite.com/feed/`) يمكن للقراء الوصول إليه.
- اختبار الخلاصة: عند زيارة الرابط، سترى ملف XML يحتوي على المحتوى، أو يمكنك استخدام برنامج مثل Feedly للاشتراك.

مثال عملي:

لنفترض أن لديك مدونة تحتوي على منشورات، وتريد إنشاء خلاصة RSS لآخر 10 منشورات.

1. إنشاء كلاس للـ Feed:

في ملف `feeds.py` داخل تطبيقك في Django، اكتب الكود التالي:

```
python
from django.contrib.syndication.views import Feed
from django.urls import reverse
from .models import Post

class LatestPostsFeed(Feed):
    title = "مدونتي - آخر المنشورات"
    link = "/feed/"
    description = "آخر التحديثات من مدونتي"

    def items(self):
        # إرجاع آخر 10 منشورات
        return Post.objects.order_by('-published_date')[:10]

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

    def item_description(self, item):
        return item.content[:100] # أول 100 حرف من المحتوى

    def item_link(self, item):
        return reverse('post_detail', args=[item.slug])
```

شرح الكود:

- `title, link, description`: هذه معلومات عامة عن الخلاصة (مثل اسم المدونة ووصلتها).
- `items`: يحدد المنشورات التي ستظهر في الخلاصة (هذا يمثل آخر 10 منشورات).
- `item_title, item_description, item_link`: تحدد التفاصيل لكل منشور (العنوان، الوصف، ورابط المنشور).

2. إعداد URL:

في ملف `urls.py` الخاص بالتطبيق، أضف رابطًا للخلاصة:

python

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

urlpatterns = [
    path('feed/', LatestPostsFeed(), name='post_feed'),
]
```

3. اختبار الخلاصة:

- افتح المتصفح واذهب إلى `./http://yourwebsite.com/feed`
- سترى ملف XML يحتوي على آخر 10 منشورات بتنسيق .RSS
- يمكنك نسخ الرابط وإضافته إلى برنامج مثل Feedly لتجربة الاشتراك.

ماذا يرى المستخدم؟

عندما يشترك شخص في الخلاصة باستخدام برنامج، سيتلقى إشعاعًا في كل مرة تنشر فيها منشورًا جديداً، مع عنوان المنشور وجزء من المحتوى. إذا نقر على المنشور، سيتم توجيهه إلى موقعك.

لماذا هذا مفيد؟

- يسهل على القراء متابعة تدوينتك.
- يزيد من التفاعل مع موقعك.
- يعطي انطباعاً احترافياً لمدونتك.

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

الهدف:

نريد إنشاء خلاصة RSS (مثل تطبيقات آخر المنشورات في مدونة) باستخدام Django. هذه الخلاصة ستسمح للقراء بمتابعة آخر منشورات مدونتك تلقائياً عبر برامج مثل Feedly.

الخطوات:

1. إنشاء ملف جديد:

- قم بإنشاء ملف باسم `feeds.py` في مجلد تطبيق المدونة الخاص بك.
- هذا الملف سيحتوي على الكود الذي ينشئ الخلاصة.

`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 LatestPostFeed(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.

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.

شرح الكود ببساطة:

1. الاستيرادات (Imports):

- `markdown` إلى (مثل النصوص المنسقة Markdown مكتبة لتحويل النصوص المكتوبة بصيغة HTML).
- `Feed` أداة من `Django` لإنشاء الخلاصات.
- `truncatewords_html` (مثل الوصف) بحيث لا يتجاوز عدداً معيناً من الكلمات.
- `reverse_lazy`: بشكل آمن حتى لو لم يتم تحميل إعدادات المشروع بعد URL أداة لإنشاء روابط.
- `Post`: المنشورات في مدونتك (Model) نموذج.

2. إنشاء كلاس الخلاصات (LatestPostFeed):

- هذا الكلاس يحدد شكل الخلاصات.
- `title = "My blog"` : عنوان الخلاصات (يظهر كاسم المدونة في برامج القراءة).
- `link = reverse_lazy('blog:post_list')` : رابط المدونة الرئيسي (مثل صفحة قائمة المنشورات).
- `description = "New posts of my blog"` : وصف مختصر للخلاصات.

3. تحديد المنشورات (items):

- هذه الدالة تختار المنشورات التي ستظهر في الخلاصات: `items(self)`.
- `Post.published.all()[:5]`: من نموذج `Post` (5 منشورات منشورة فقط).

4. تفاصيل كل منشور:

- تعيين عنوان المنشور (مثل "أول منشور في مدونتي") `item_title(self, item)`.
 - تعيين وصفاً قصيراً للمنشور: `item_description(self, item)`.
 - `markdown.markdown(item.body)`: إلى Markdown تحويل نص المنشور من.
 - `truncatewords_html(..., 30)`: HTML تقص النص ليصبح 30 كلمة فقط، مع الحفاظ على تنسيق.
- صحيح.
- `item_pubdate(self, item)`: تعيين تاريخ نشر المنشور (مثل "29-04-2025").

ماذا يحدث عند تشغيل الكود؟

- يحتوي على آخر 5 منشورات (العنوان، الوصف المختصر، تاريخ النشر، والرابط) XML ينشئ ملف `Django`.
- يمكن للقراء الوصول إلى هذا الملف عبر رابط `yourwebsite.com/feed/`.
- سينتقلون إلى منشورات بكل منشور جديد، Feedly إذا أضافوا الرابط إلى برنامج مثل.

لماذا ؟ `reverse_lazy`

- قبل إنشاء الرابط (URLs) لأنها يتطلب حتى يتم تحميل إعدادات المشروع `reverse_lazy` يستخدم بدلاً من.
- هذا مفید لأن الخلاصات قد تُحمل قبل تحميل كل الإعدادات.

لماذا ؟ `truncatewords_html`

- إذا كان وصف المنشور طويلاً (مثل 1000 كلمة)، لا نريد إظهاره كاملاً في الخلاصات.
- هذه الأداة تقص النص إلى 30 كلمة فقط، وتتأكد أن HTML (مثل الوسوم `<p>`) لا تتكسر.

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 LatestPostFeed
```

```

app_name = 'blog'

urlpatterns = [
    # post views
    path("", views.post_list, name='post_list'),
    path('tag/<slug:tag_slug>/', views.post_list, name='post_list_by_tag'),
    # path("", views.PostListView.as_view(), name='post_list'),
    # path('<int:id>', views.post_detail, name='post_detail'),
    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/', LatestPostFeed(), 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>Tue, 02 Jan 2024 16:30:00 +0000</lastBuildDate>
        <item>
            <title>Markdown post</title>
            <link>http://localhost:8000/blog/2024/1/2/markdown-post/</link>
            <description>This is a post formatted with ...</description>
            <guid>http://localhost:8000/blog/2024/1/2/markdown-post/</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:



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



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:

Sources Groups Rules Service Preferences About

OPML File

Import Export

Add source

http://127.0.0.1:8000/blog/feed/

Add

Name	URL
My blog	http://127.0.0.1:8000/blog/feed/

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:

The screenshot shows the Fluent Reader application interface. At the top, there are window control buttons (red, yellow, green) and a title bar with 'All articles'. Below the title bar, there are five cards representing blog posts:

- Markdown post**
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 ...
- Notes on Duke Ellington**
Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...
- Who was Miles Davis?**
Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.
- Who was Django Reinhardt?**
Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...
- Another post**
Post body.

Click on a post to see a description:

My blog

Notes on Duke Ellington

1/2/2024, 5:00:00 PM

Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...

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

Notes on Duke Ellington

1/2/2024, 5:00:00 PM

Published Jan. 2, 2024, 4 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century.

[Share this post](#)

Similar posts

[Who was Miles Davis?](#)

[Who was Django Reinhardt?](#)

0 comments

There are no comments yet.

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:

```
#####
<div class="card p-3">
    <h2 class="fw-bold">My Blog</h2>
    <p>Welcome to my blog! Explore various topics and enjoy reading.</p>
    <p>I've written {% total_posts %} posts so far.</p>
    <p>
        <a href="{% url 'blog:post_feed' %}">Subscribe to my RSS feed
    </a>
    </p>
    <h4 class="text-primary">Latest posts</h4>
    {% show_latest_posts 3 %}
</div>
```

```
<h4 class="text-primary">Most commented posts</h4>

{% get_most_commented_posts as most_commented_posts %}

<ul>

    {% for post in most_commented_posts %}

        <li>

            <a style="color: black;" href="{{ post.get_absolute_url }}>{{ post.title }}</a>

        </li>

    {% endfor %}

</ul>

</div>
```

#####

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:

My blog

This is my blog. I've written 5 posts so far.

[Subscribe to my RSS feed](#)

Latest posts

- [Markdown post](#)
- [Notes on Duke Ellington](#)
- [Who was Miles Davis?](#)

Most commented posts

- [Who was Django Reinhardt?](#)
- [Who was Miles Davis?](#)
- [Notes on Duke Ellington](#)
- [Markdown post](#)

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 how important different fields are (for example, the 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 database 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.

Note: 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.

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.

Note: 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.

PostgreSQL provides a Docker image that makes it very easy to deploy a PostgreSQL server with a standard configuration.

1. ليش نحتاج خاصية البحث النصي الكامل؟

في أي موقع ويب، خصوصاً المدونات، المستخدمين يحتاجوا بحثاً عن محتوى معين (مثل مقالات تحتوي على كلمة مينة)، لو عايز تبحث في قاعدة البيانات عن كلمة مينة، ممكن تستخدم أمر بسيطة في Django زي:

python

... ⌂ Copy

```
Post.objects.filter(body__contains='framework')
```

هذا الأمر يبحث في تصووص المقالات (في حقل `body`) عن الكلمة "framework". لكن هذه الطريقة:

- **بسطّة جداً**: يبحث عن الكلمة بشكل حرفي (يعني لو الكلمة مثلاً مكتوبة بنفس الطريقة، مارح تلاقّها).
- **مش دقيقه**: ما ينطّلّك النتائج مرتّبة حسب الأهمية (مثلاً، لو الكلمة موجودة في العنوان، هي أهم من وجودها في النص).
- **مش قويّة**: لو النص طويّل أو عنده كمية بيانات كبيرة، مارح تكون فعالة.

عندن كده، إذا كنت عايز بحث أكثر تكاء، زي:

- إيجاد كلمات مشابهة (مش بس مطابقة حرفية).
- ترتيب النتائج حسب الأهمية (مثلاً، الكلمة في العنوان أهم من النص).
- التعامل مع تصووص طويّلة بكتفاه. هنا بحتاج إلى **محرك بحث نصي كامل** (Full-Text Search Engine).

2. ليش هو البحث النصي الكامل؟

البحث النصي الكامل هو طريقة ذكية للبحث في التصووص. بدل ما يبحث عن كلمة بشكل حرفـي، المحرك:

- يفحص الكلمات نفسها (مش بس حروف).
- بيقدر يربّب النتائج حسب مدى ارتباطها بالبحث.
- بيقدر يتعامل مع كلمات مشابهة أو متـابقات.
- بيستخدم تقنيات مثل وزن الكلمات (يعني يعطي أهمية أكبر للكلمـة لو موجودـة في العنوان مقارنة بالنص العادي).

3. Django الكامل والبحث النصي

لأن لإضافة خاصية البحث (النحو، PostgreSQL، MySQL، SQLite)، كإطار عمل يحتوي مع قاعدة بيانات مختلفة Django يتوفر أدوات خاصية لقاعدة بيانات PostgreSQL، django.contrib.postgres.

أمثلة PostgreSQL

- قاعدة بيانات قوية جدًا وألها دعم دمج للبحث النصي الكامل.
- يتوفر ميزات متقدمة رى البحث عن كلمات بطريقة ذكية.
- ترتيب النتائج حسب الأهمية.
- التعامل مع كميات بيانات كبيرة بكفاءة.
- وهي وحدة مخصصة لدعم ميزات django.contrib.postgres، بفضل هذه الميزات من خلال وحدة PostgreSQL، الذي من موجودة في قاعدة بيانات تابية PostgreSQL.

أمثلةSQLite

مذرووث حالياً يستخدم قاعدة بيانات SQLite، وهي:

- عملية وسهلة جدًا للتطوير (يعني لما تكون بتجرب الأشياء على جهازك).
- لكن دعمها للبحث النصي الكامل محدود جدًا، وDjango ما يتوفر أدوات جاهزة لدعم هذه الميزة مع SQLite.

مثمن كده، إذا كنت علين مستخدم البحث النصي الكامل، لازم:

1. تحول قاعدة البيانات من SQLite إلى PostgreSQL.
2. تستخدم وحدة django.contrib.postgres في Django لاستغلال ميزات البحث النصي.

4. أمثلة SQLite كافية للتطوير بس مش للاحتياج؟

• SQLite:

- بسيطة وسهلة الإعداد (ما تحتاج سيرفر منفصل).
- ملائمة للتجربة والتطوير لأنها عملية.
- لكنها من قوية كافية للتعامل مع:
- كميات بيانات كبيرة.
- عدد كبير من المستخدمين.
- ميزات متقدمة رى البحث النصي الكامل.

• PostgreSQL أو MySQL أو MariaDB):

- قاعدة بيانات قوية مصممة للاحتياج (يعني لما تنشر موقعك على الإنترنت).
- يدعم ميزات متقدمة رى البحث النصي، الأداء العالي، والتوزيع (Scalability).

Installing Docker

Docker is a popular open-source containerization platform. It enables developers to package applications into containers, simplifying the process of building, running, managing, and distributing applications.

First, download and install Docker for your OS. You will find instructions for downloading and installing Docker on Linux, macOS, and Windows at <https://docs.docker.com/get-docker/>. The installation includes both Docker Desktop and Docker command-line interface tools.

Installing PostgreSQL

After installing Docker on your Linux, macOS, or Windows machine, you can easily pull the PostgreSQL Docker image. Run the following command from the shell:

```
docker pull postgres:16.2
```

This will download the PostgreSQL Docker image to your local machine.

ما هو Docker ولماذا نستخدمه؟

الحاوية تشبه آلية افتراضية خفيفة تحتوي على كل ما يحتاجه **Containers** هو أداة تسمح بتشغيل التطبيقات داخل **حاويات Docker** هو عملية قد تكون معقدة وتحتاج حسب نظام التشغيل، Docker يجعل العملية بدلًا من تثبيت PostgreSQL يدوياً على جهازك (وهو عملية قد تكون معقدة وتحتاج حسب نظام التشغيل)، Docker يجعل العملية:

لماذا نستخدم Docker لتنصيب PostgreSQL؟

بدلاً من تثبيت PostgreSQL يدوياً على جهازك (وهو عملية قد تكون معقدة وتحتاج حسب نظام التشغيل)، Docker يجعل العملية:

1. سهلة: تقوم بتنزيل صورة (Image) (جاهزة) بأمر واحد.
2. موحدة: تجعل بنفس الطريقة على Windows، macOS، Linux، أو macOS.
3. نظيفة: الحاوية معزلة عن نظامك، فلا تؤثر على إعدادات جهازك.
4. مناسبة للإنتاج: يمكن استخدام نفس الصورة في بيئة التطوير والإنتاج.

كيف يتم استخدام Docker في هذا المشروع؟

إضافة البحث الكامل إلى المدونة، تحتاج إلى تشغيل خادم PostgreSQL. بدلاً من تثبيته يدوياً، سنتستخدم Docker لتحميل صورة PostgreSQL ويشتغلها كحاوية. إليك الخطوات بالتفصيل:

1. تثبيت Docker

- قم بتنزيل وتثبيت Docker على جهازك من موقع Docker الرسمي.
- التنصيب يشمل:

- واجهة رسومية لإدارة الحاويات.
- Docker Desktop: Docker CLI: أداة سطر أوامر لتنشيل الأوامر.

2. تحميل صورة PostgreSQL

افتح المتصفح (Terminal) وشغّل الأمر التالي:

```
bash ... ⌂ Copy
```

```
docker pull postgres:16.2
```

- هذا الأمر يقوم بتحميل صورة PostgreSQL (الإصدار 16.2) من متجر Docker Hub.
- الصورة هي عبارة عن قالب جاهز يحتوي على PostgreSQL وجميع الإعدادات الأساسية.

3. تشغيل حاوية PostgreSQL

بعد تحميل الصورة، يمكنك تشغيل حاوية PostgreSQL باستخدام الأمر التالي:

```
bash ... ⌂ Copy
```

```
docker run --name my-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres
```

شرح الأمر:

- name my-postgres: يعطي اسمًا للحاوية (يمكنك تغييره).
- e POSTGRES_PASSWORD=mysecretpassword: يحدد كلمة مرور لقاعدة البيانات.
- d: يشغّل الحاوية في الخلفية.
- p 5432:5432: يربط المنفذ الافتراضي 1) على جهازك بالمنفذ 5432 داخل الحاوية PostgreSQL.
- postgres: اسم الصورة التي سيتم تشغيلها.

Execute the following command in the shell to start the PostgreSQL Docker container:

```
docker run --name blog_db -e POSTGRES_DB=blog -e POSTGRES_USER=blog -e POSTGRES_PASSWORD=xxxxx -p 5432:5432 -d postgres:16.2
```



شرح الأمر المصحح

يُنشئ هذا الأمر حاوية جديدة ويبدا تشغيلها من صورة محددة: `docker run`.

للحاوية لسهولة الرجوع إليه `blog_db`: يُعين اسم `--name blog_db`.

عند تشغيل `blog` يُنشئ هذا قاعدة بيانات باسم `blog`. يُعين اسم قاعدة البيانات إلى: `-e POSTGRES_DB=blog`.

عند تشغيل `blog` يُنشئ اسم المستخدم لقاعدة بيانات `blog`: `-e POSTGRES_USER=blog`.

عند تشغيل `blog` يُعين كلمة مرور مستخدم PostgreSQL 281133: `-e POSTGRES_PASSWORD=281133`.

المنفذ الافتراضي (ل) يُعين المنفذ 5432 على جهازك المحلي إلى المنفذ 5432 داخل الحاوية: `-p 5432:5432`.

يُشغل الحاوية في وضع التشغيل المنفصل (في الخلفية): `-d`.

يُحدد صورة PostgreSQL 16.2: `postgres:16.2`.

This command starts a PostgreSQL instance. The `--name` option is used to assign a name to the container, in this case, `blog_db`. The `-e` option is to define environment variables for the instance. We set the following environment variables:

- `POSTGRES_DB`: Name of the PostgreSQL database. If not defined, the value of `POSTGRES_USER` is used for the database name.
- `POSTGRES_USER`: Used in conjunction with `POSTGRES_PASSWORD` to define a username and password. The user is created with superuser power.
- `POSTGRES_PASSWORD`: Sets the superuser password for PostgreSQL.

شرح النص

النص يصف كيفية تشغيل خالم PostgreSQL داخل حاوية Docker باستخدام الأمر `docker run`، مع تفاصيل الخيارات المستخدمة لتهيئة قاعدة البيانات، كما يوضح كيفية تثبيت مكتبة PostgreSQL Python لربط Django بـ PostgreSQL. وينظر الخطوة التالية وهي نقل البيانات من قاعدة SQLite إلى PostgreSQL.

1. الأمر الأساس: تشغيل حلبة PostgreSQL

الأمر المستخدم هو:

- **docker run** :
 - **--name blog_db** :
 - **-e POSTGRES_DB=blog** :
 - **-e POSTGRES_USER=blog** :
 - **-e POSTGRES_PASSWORD** :

- **-e POSTGRES_USER=blog :**

الفاكهة: يحدد اسم المستخدم الذي سيتم إنشاؤه في PostgreSQL (هذا `blog`). هذا المستخدم يكون موصى به مُزدوج (Superuser)، مما يعني أنه يملك كامل الصلاحيات لإدارة قاعدة البيانات.

يُستخدم هذا المستخدم للاتصال بقاعدة البيانات من Django أو أدوات أخرى مثل `psql`.
- **-e POSTGRES_PASSWORD=281133 :**

الفاكهة: يحدد كلمة المرور للمستخدم المنشأ (هذا `281133`). بدون كلمة مرور، لن يسمح PostgreSQL بتشغيل الحاوية لأسباب أمنية.

هذه الكلمة سُتُستخدم لاحقًا في إعدادات Django للاتصال بقاعدة البيانات.
- **-p 5432:5432 :**

الفاكهة: يربط المنفذ `5432` على جهازك الم المحلي (Host) بالمنفذ `5432` داخل الحاوية. المنفذ `5432` هو المنفذ الافتراضي الذي يُعمل عليه PostgreSQL.

هذا يسمح لتطبيقات خارج الحاوية (مثل Django على جهازك) بالاتصال بقاعدة البيانات باستخدام `localhost:5432`.

بدون هذا الخيار، لن تتمكن التطبيقات الخارجية من الوصول إلى قاعدة البيانات.
- **-d :**

الفاكهة: يشغل الحاوية في الوضع المنفصل (Detached Mode)، أي في الخلفية. هذا يعني أن المترمين (Terminal) لن يتعطل مشغولة بعرض مخرجات الحاوية، ويمكنك متابعة العمل.

بدون هذا الخيار، سترى سجلات تشغيل PostgreSQL مباشرةً في الطريقة، وقد تحتاج إلى فتح طرقية أخرى لمواصلة العمل.
- **postgres:16.2 :**

الفاكهة: يحدد صورة Docker التي سيتم استخدامها (الإصدار 16.2). هذه الصورة تحتوي على كل ما يلزم PostgreSQL تشغيل خادم.

يُتم تحميلها مسبقًا باستخدام `docker pull postgres:16.2`.

The `-p` option is used to publish the 5432 port, on which PostgreSQL runs, to the same host interface port. This allows external applications to access the database. The `-d` option is for detached mode, which runs the Docker container in the background.

Open the Docker Desktop application. You should see the new container running, as in *Figure 3.27*:

	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
	 blog_db cee60bc94f4d	postgres	Running	0.04%	5432:5432	42 seconds ago	 

You will see the newly created `blog_db` container, with the status `Running`. Under `Actions`, you can stop or restart the service. You can also delete the container. Note that deleting the container will also eliminate the database and all the data it contains.

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

```
python -m pip install psycopg==3.1.18
```

2. نتيجة تشغيل الأمر

بعد تنفيذ الأمر أعلاه يندرج:

- سيتم إنشاء حاوية تسمى `blog_db` وتشغيلها.
- يمكنك التحقق من أن الحاوية تعمل باستخدام:

```
bash ... ⌂ Copy
docker ps
```

سترى `blog_db` مع حالة `Running` (جاري التشغيل).

إذا كنت تستخدم **Docker Desktop**, يمكنك رؤية الحاوية في الواجهة السومية، حيث يمكنك:

- إيقاف الحاوية (Stop).
- إعادة تشغيلها (Restart).
- حذفها (Delete). ملاحظة: حذف الحاوية سيؤدي إلى حذف قاعدة البيانات وجميع البيانات المخزنة فيها، ما لم تكن قد حفظت البيانات في وحدة تخزين دائمة (Volume).

3. تثبيت مكتبة `psycopg`

النص يذكر الأمر التالي لتنصيب محول `Python` لـ `PostgreSQL`:

```
bash ... ⌂ Copy
python -m pip install psycopg==3.1.18
```

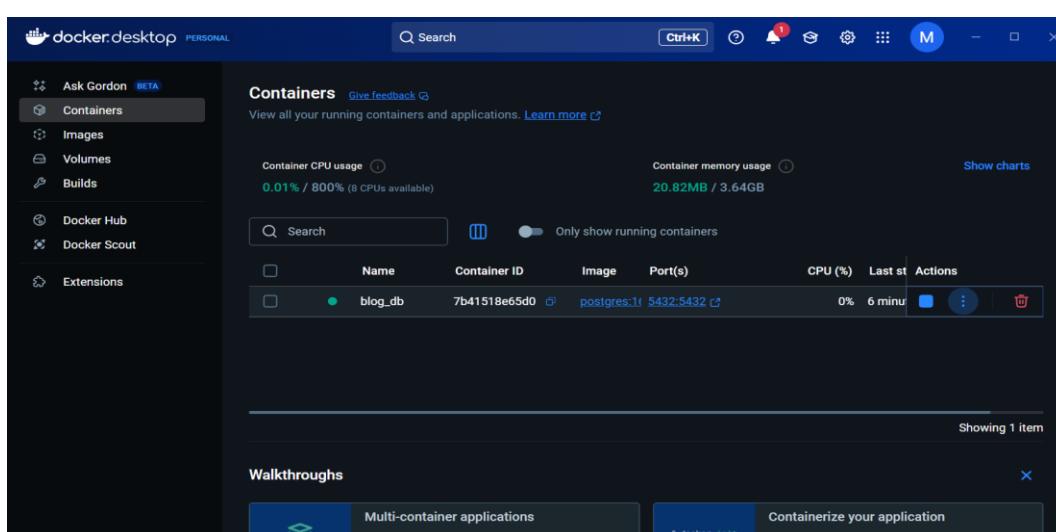
النتيجة:

- مكتبة `psycopg` هي المحول (Adapter) الذي يسمح `Django` بالاتصال مع قاعدة بيانات `PostgreSQL`.
- بدون هذه المكتبة، لن يمكن `Django` من الاتصال بقاعدة البيانات.
- الإصدار المحدد (3.1.18) يضمن التوافق مع `Django` و `PostgreSQL`.

كيف يعمل:

- الأمر يستخدم `pip` (مدير الحزم في `Python`) لتنصيب المكتبة داخل بيئة `Python` الافتراضية (Virtual Environment) التي تعمل فيها (كما هو واضح من `(venv)` في موجه الأوامر).

Next, we will migrate the existing data in the SQLite database to the new PostgreSQL instance.



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 format. 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
```

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.

تقسيم النص وشرحه بشكل مفصل

1. الهدف العام: تصدير البيانات قبل تبديل قاعدة البيانات
النص:

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

الشرح:

• الهدف:

الهدف هو نقل كل البيانات الموجودة في قاعدة البيانات الحالية (SQLite) إلى قاعدة بيانات جديدة (PostgreSQL). هذا ضروري لأن SQLite محدودة في دعم ميزات متقدمة مثل البحث الكامل، بينما PostgreSQL توفر ميزات قوية مثل

`django.contrib.postgres.tesquery` عبر وحدة

• لماذا نحتاج إلى تصدير البيانات؟

عد تغيير قاعدة البيانات في Django، فإن البيانات (مثل المقالات، المستخدمين، التحليقات) لا تتغافل طقائياً من SQLite إلى PostgreSQL، لأن كل قاعدة بيانات هي كيًّاً مسبقاً. لذلك، يجب تصدير البيانات من SQLite، ثم استيرادها إلى PostgreSQL.

• الخطوات الأساسية:

1. تصدير البيانات (Dump): استخراج البيانات من SQLite إلى ملف.

2. تبديل قاعدة البيانات: تغيير إعدادات Django PostgreSQL لاستخدام PostgreSQL.

3. استيراد البيانات (Load): إدخال البيانات المصدرة إلى PostgreSQL.

• الميزة:

هذه العملية جزء من الانتقال إلى PostgreSQL لاستغادة من ميزات البحث الكامل، التي تتطلب قاعدة بيانات قوية مثل PostgreSQL.

فائدة:

- تضمن استمرارية البيانات (مثل مقالات المدونة) عند التحول إلى قاعدة بيانات جديدة، مما يمنع فقدان المعلومات.
- تمكنك من الاستغادة من ميزات PostgreSQL دون الحاجة إلى إعادة إدخال البيانات يدوياً.

لـ حمل:

مثـل صـلـي:

إذا كان لديك مدونة تحتوي على 50 مقالة في SQLite، فإن تصدير البيانات يضمن نقل هذه المقالات إلى PostgreSQL لاستغادة في الموقع.

2. مفهـوم فـixtـures فـي Django

النص:

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 format. We are going to create a fixture ".with all data contained in the database

الشرح:

• ما هي fixtures؟

- فـيـاسـها Django يمكن لـ (JSON أو XML أو YAML) هـيـ مـلـفـاتـ تحـتـويـ عـلـىـ بـيـانـاتـ قـاعـدـةـ بـيـانـاتـ بـصـيـغـةـ مـدـظـمـةـ Fixtures أو كـلـيـاتـهاـ أوـ كـلـيـاتـهاـ
- ـ يـتـعـدـمـ لـ أـمـرـاـتـ مـلـىـ

- نقل البيانات بين قواعد بيانات مختلفة (كما في حالات من SQLite إلى PostgreSQL).
- توفير بيانات أولية (Initial Data) لتطبيق جديد.
- اختبار التطبيقات ببيانات وهمية.
- **الصيغة المدعومة:**
 - الصيغة الافتراضية، حقيقة، سهلة القراءة، ومدعومة على نطاق واسع: **JSON**: أقل شرطًا، تُستخدم في بعض الأنظمة القديمة.
 - **XML**: صيغة مرونة، لكنها أقل استخدامًا في Django.
- **الهدف هنا:** سلقى ملف **Fixture** بصيغة **JSON** يحتوي على **كل البيانات** في قاعدة **SQLite**، بما في ذلك بيانات النماذج مثل **Post** ، **User** ، **...** وغيرها.
- **الملفات JSON؟**
- التعامل معها يسهلة عند **Django** هي الصيغة الافتراضية لأنها متوافقة مع معظم الأنظمة، سهلة القراءة، ويمكن لـ **JSON** **loaddata** (والاستيراد) **dumpdata** (التصدير) .
- **هيكل Fixture:** ملف **JSON** الناتج سيكون قائمة من الكائنات، كل كائن يمثل سجلًا في قاعدة البيانات، مع تفاصيل النموذج، المفتاح الاسمي، والحقول.
- **فائدة:**
 - توفر طريقة موحدة وسهلة لتصدير واستيراد البيانات بين قواعد البيانات.
 - تسمح بحفظ نسخة احتياطية من البيانات يمكن استخدامها لاحقًا.
 - تسهل نقل البيانات دون الحاجة إلى كتابة سكريبتات محددة.

3. الأمر **dumpdata** ووظيفته

العنوان:

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

الشرح:

- **ما هو الأمر **dumpdata**؟**
- هو أمر إداري في **Django** يستخدم لتصدير (Dumping) بيانات قاعدة البيانات إلى:
 - الإخراج القياسي (Standard Output)، أي الطريقة بشكل افتراضي.
 - أو إلى ملف إذا تم تحديده.
- يحول البيانات إلى صيغة **JSON** بشكل افتراضي، مما يجعلها جاهزة لاستيراد باستخدام الأمر **loaddata** .
- **ماذا يحتوي الإخراج؟**
- الإخراج عبارة عن هيكل بيانات **JSON** يحتوي على:
 - **اسم النموذج (Model):** مثل **auth.user** أو **blog.post** .
 - **المفتاح الأساسي (Primary Key):** لتحديد كل سجل بشكل فريد.
 - **الحقول (Fields):** قيم الحقول (مثل العنوان، المحتوى، التاريخ).
- هذا الهيكل ضروري لأن **Django** يحتاج إلى هذه المعلومات لإعادة إنشاء البيانات في قاعدة بيانات أخرى.
- **لماذا هذا مهم؟**
- الهيكل يضمن أن **Django** يمكنه التعرف على النماذج والسجلات عدد الاستيراد، مما يحافظ على سلامة البيانات.
- على سبيل المثال، يحتاج **Django** إلى معرفة أن سجل معين ينتمي إلى نموذج **Post** وليس **User** .

فائدة الأمر **dumpdata:**

- يوفر طريقة بسيطة وسريعة لتصدير كل البيانات أو جزء منها من قاعدة البيانات.
- ينتج ملفًا (Fixture) يمكن استخدامه لنقل البيانات أو الاحتفاظ به كنسخة احتياطية.
- يدعم تخصيص الإخراج (مثل تحديد تطبيقات أو نماذج محددة).

- تخصيص الإخراج حسب التطبيقات أو الملفات:

- يمكنك تخصيص بيانات تطبيق معين (App) أو نموذج معين (Model) بدلاً من تخصيص كل شيء.

- المثلاً:

- تخصيص بيانات تطبيق blog فقط:

```
bash ... ⌂ Copy
python manage.py dumpdata blog
```

فأمثلة: يظل حجم الملف إذا كنت لا تحتاج إلى بيانات تطبيقات أخرى (مثل auth).

- تخصيص نموذج Post فقط:

```
bash ... ⌂ Copy
python manage.py dumpdata blog.Post
```

فأمثلة: يرتكز على بيانات نموذج معين، مما يوفر الوقت إذا كنت تهتم بجزء محدد.

- تغيير الصيغة باستخدام --format--:

- بشكل افتراضي، الصيغة هي JSON، لكن يمكنك تحديد صيغة أخرى:

```
bash ... ⌂ Copy
python manage.py dumpdata --format=xml
```

أو

```
bash ... ⌂ Copy
python manage.py dumpdata --format=yaml
```

فأمثلة: يتيح استخدام صيغة أخرى إذا كنت مطلوبة لتنظيم معين، لكن JSON هو الأفضل عموماً.

- حفظ الإخراج في ملف باستخدام --output--:

- بدلاً من عرض البيانات في الطريقة التقليدية، يمكنك حفظها في ملف:

```
bash ... ⌂ Copy
python manage.py dumpdata --output=mysite_data.json
```

فأمثلة: ينشئ مثلاً دليلاً يمكن نقله أو استخدامه لاحقاً للاستيراد.

- تحسين القراءة باستخدام --indent--:

- الخيار indent=2 يضيف مسافات بادئة (Indentation) لجعل ملف JSON أكثر وضوحاً:

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

فأمثلة: يجعل الملف مفروعاً للإنسان، مما يسهل فحصه أو تحريره يدوياً إذا لزم الأمر.

- بدون indent، تكتب البيانات في سطر واحد، مما يجعلها صعبة القراءة.

- معرفة المزيد باستخدام --help--:

- الأمر التالي يعرض جميع الخيارات المتاحة لـ dumpdata:

```
bash ... ⌂ Copy
python manage.py dumpdata --help
```

فأمثلة: يساعدك على اكتشاف خيارات إضافية مثل:

- --exclude: لاستبعاد تطبيقات أو نماذج.

- --natural-foreign: لاستخدام مفاهيم خارجية طبيعية.

- --all: لتصدير كل البيانات بما في ذلك البيانات المدارسة.

```
الامر المستخدم: python manage.py dumpdata --indent=2 --output=mysite_data.json
• تفاصيل الأمر:
• يصدر كل البيانات من قاعدة SQLite، بما في ذلك بيانات التطبيقات المبنية مثل blog، auth (المستخدمين)، contenttypes، الخ.
• يحفظ البيانات في ملف يسمى mysite_data.json في المجلد الحالي (في حالك: C:\Users\mialj\Desktop\blog\src).
• يستخدم --indent=2 لتنسيق الملف بشكل مقرئ.
• فلدة الأمر:
• ينتهي ملف JSON بحتوي على نسخة كاملة من قاعدة البيانات، جاهزة لاستيراد إلى PostgreSQL.
• يوفر وسيلة موثوقة لنقل البيانات أو الاحتفاظ بها كنسخة احتياطية.
• النتيجة:
• يتم إنشاء ملف mysite_data.json يحتوي على هيكل JSON يصف كل سجل في قاعدة البيانات.
• يمكنك فتح الملف بمحرر نصوص (مثل VS Code) لرؤى البيانات، التي تتضمن كل تموذج (مثل blog.Post، auth.User) مع حقوله.
• محتوى الملف:
• يحتوي على قائمة من الكائنات، كل كائن يمثل سجلًا، مع:


- "model": اسم التموذج (مثل blog.post).
- "pk": المفتاح الأساسي.
- "fields": قيم الحقول (مثل العنوان، المحتوى).

```

```
json
```

```
[  
  {  
    "model": "blog.post",  
    "pk": 1,  
    "fields": {  
      "title": "مدخل إلى Django",  
      "body": "شرح من كيفية استخدام Django.",  
      "created": "2025-05-04T10:00:00Z"  
    }  
  },  
  {  
    "model": "auth.user",  
    "pk": 1,  
    "fields": {  
      "username": "admin",  
      "email": "admin@example.com"  
    }  
  }  
]
```

كيفية التحقيق؟

تحقق من وجود الملف.

```
powershell
```

```
dir mysite_data.json
```

- فتح الملف للتأكد من أن البيانات صحيحة ومحسنة.
- فائدة هذا الأمر المقترن.
- يرفع سمة كاملة ومنظمة من بيانات SQLite قادمة في ملف **.JSON**.
- يجعل البيانات جاهزة لاستيراد إلى **PostgreSQL**.
- التنسيق المقترن (`indent=2`) يسهل قصص الملف أو تغييره إذا لزم الأمر.

٦. التعامل مع أخطاء الترميز (Encoding Errors)

النص:

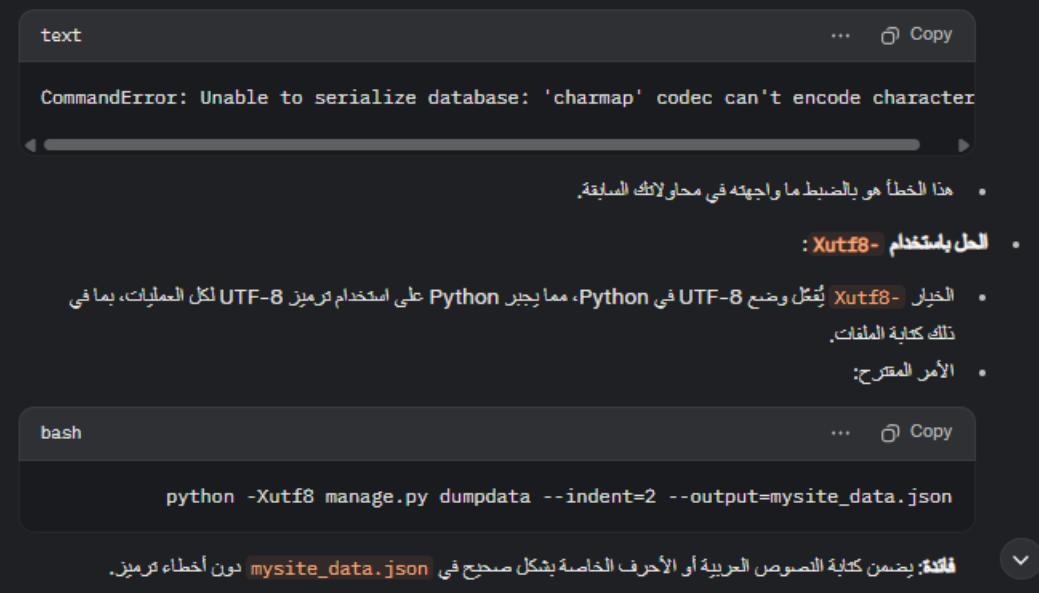
If you get an encoding error when running the command, include the `-Xutf8` flag as follows to "activate Python UTF-8 mode

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

الشرح:

• ما هي مشكلة الترميز؟

- إذا كانت قاعدة البيانات تحتوي على أحرف غير ASCII (مثل نصوص عربية، إيموجي، أو رموز خاصة)، فقد يفشل الأمر `dumpdata` على Windows.
- السبب: Windows يستخدم ترميزاً افتراضياً محدوداً (`cp1252` أو `charmap`) لا يدعم الأحرف غير اللاتينية، مما يؤدي إلى خطأ متن:



text ... ⌂ Copy

```
CommandError: Unable to serialize database: 'charmap' codec can't encode character
```

هذا الخطأ هو بالضبط ما واجهته في محاولةك السابقة.

• **الحل باستخدام `-Xutf8`:**

- الخيار `-Xutf8` يُفعّل وضع UTF-8 في Python، مما يجبر Python على استخدام ترميز UTF-8 لكل العمليات، بما في ذلك كتابة الملفات.
- الأمر المفترض:

bash ... ⌂ Copy

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

فأنت يضمن كتابة النصوص العربية أو الأحرف الخاصة بشكل صحيح في `mysite_data.json` دون أخطاء ترميز.

• متى تستخدم هذا الحل؟

- عند ظهور خطأ ترميز، خاصة إذا كانت بريئة تحتوي على نصوص عربية أو أحرف غير ASCII.
- في حالة، واجهت هذا الخطأ، لذا هذا الحل ضروري.

مثال حللي:

إذا كانت قاعدتك تحتوي على مقالة بعنوان "مدخل إلى Django" (نص عريبي)، فإن تشغيل الأمر بدون `Xutf8` سيفشل، باستخدام `PYTHONIOENCODING` أو `Xutf8` سيمتصّر النص العربي بشكل صحيح.

فأنت الأمر مع `-Xutf8`:

- يحل مشكلة الترميز على Windows، مما يسمح بتصدير الأحرف غير اللاتينية.
- يضمن أن ملف `mysite_data.json` يحتوي على البيانات كاملة وصحيحة.

Switching the database in the project

Now you will add the PostgreSQL database configuration to your project settings.

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': config('DB_NAME'),  
        'USER': config('DB_USER'),  
        'PASSWORD': config('DB_PASSWORD'),  
        'HOST': config('DB_HOST'),  
    }  
}
```

The database engine is now postgresql. The database credentials are now loaded from environment variables using python-decouple.

Let's add values to the environment variables. Edit the .env file of your project and add the following lines highlighted in bold:

```
EMAIL_HOST_USER=your_account@gmail.com  
EMAIL_HOST_PASSWORD=xxxxxxxxxxxx  
DEFAULT_FROM_EMAIL=My Blog  
DB_NAME=blog  
DB_USER=blog  
DB_PASSWORD=xxxxxx  
DB_HOST=localhost
```

Replace xxxxxx with the password you used when starting the PostgreSQL container. The new data base 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 taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
  Applying taggit.0005_auto_20220424_2025... OK
  Applying taggit.0006_rename_taggeditem_content_type_object_id_taggit_tagg_
content_8fc721_idx... OK
  Applying blog.0001_initial... OK
  Applying blog.0002_alter_post_slug... OK
```

The PostgreSQL database is now in sync with your data models and you can run your Django project pointing to the new database. Let's get the database to the same state by loading the data we previously exported from SQLite

Loading the data into the new database

We are going to load the data fixtures we generated previously into our new PostgreSQL database.

Run the following command to load the previously exported 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:

Select post to change					
<input type="text"/> <input type="button" value="Search"/>					
« 2024 January 1 January 2 January 3					
Action: <input type="button" value="-----"/> <input type="button" value="Go"/> 0 of 5 selected					
<input type="checkbox"/>	TITLE	SLUG	AUTHOR	PUBLISH	2 ▲ STATUS 1 ▲
<input type="checkbox"/>	Another post	another-post	admin	Jan. 1, 2024, 11:57 p.m.	Published
<input type="checkbox"/>	Who was Django Reinhardt?	who-was-django-reinhardt	admin	Jan. 1, 2024, 11:59 p.m.	Published
<input type="checkbox"/>	Who was Miles Davis?	who-was-miles-davis	admin	Jan. 2, 2024, 1:18 p.m.	Published
<input type="checkbox"/>	Markdown post	markdown-post	admin	Jan. 2, 2024, 4:30 p.m.	Published
<input type="checkbox"/>	Notes on Duke Ellington	notes-on-duke-ellington	admin	Jan. 3, 2024, 1:19 p.m.	Published

Simple search lookups

Having enabled PostgreSQL in our project, we can now build a powerful search engine by leveraging PostgreSQL's full-text search capabilities. We will begin with basic search lookups and progressively incorporate more sophisticated features, such as stemming, ranking, or weighting queries, to build a comprehensive full-text search engine.

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

```
INSTALLED_APPS = [  
    #####  
    'django.contrib.staticfiles',  
    'django.contrib.postgres',  
    'taggit',  
    'blog.apps.BlogConfig',  
]
```

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='four')  
<QuerySet [<Post: post four>]>
```

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='post')
```

```
<QuerySet [<Post: post four>, <Post: post three>, <Post: post two>, <Post: post one>]>
```

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.

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 CommentForm, EmailPostForm, 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 %}



# Posts containing "{{ query }}"



### {% with results.count as total_results %} Found {{ total_results }} result{{ total_results|pluralize }} {% endwith %}



#### {% for post in results %}


```

```

<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:

```
app_name = 'blog'

urlpatterns = [
    # post views
    #####
    path('feed/', LatestPostFeed(), 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:

Search for posts

Query:

SEARCH

Figure 3.29: 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 "jazz"

Found 3 results

[Notes on Duke Ellington](#)

Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of ...

[Who was Miles Davis?](#)

Miles Davis was an American trumpeter, bandleader, and composer. He is among ...

[Who was Django Reinhardt?](#)

Jean Reinhardt, known to all by his Romani nickname Django, was a ...

[Search again](#)

١. ما هو البحث النصي الكامل (Full-Text Search)؟

البحث النصي الكامل هو تقنية تُستخدم للبحث في النصوص داخل قاعدة البيانات بطريقة فعالة وذكية. بدلاً من البحث عن تطابق تام للكلمات (مثلاً استخدام `LIKE` في SQL)، يسمح البحث النصي الكامل بما يلي:

- البحث عن كلمات مشابهة أو متزدفات.
 - تجاهل الكلمات الشائعة (مثل "و", "في").
 - عدم البحث عن كلمات جذرية (stemming), مثل البحث عن "يجري" يطابق "جرى" أو "يجري".
 - تصنيف النتائج بناءً على مدى ارتباطها بالبحث.

في مشروعك، يتم استخدام هذه التقنية للسماح للمستخدمين بالبحث عن منشورات المدونة بناءً على العنوان أو المحتوى.

٢. لماذا نستخدم PostgreSQL لـ Opus البحث؟

بعض المزايـا PostgreSQL يـوفر مـيزـات قـوـيـةـ الـبـحـثـ النـصـيـ الكـاملـ مـقـارـنـةـ بـقـاعـدـ بـيـانـاتـ أـخـرـىـ مـثـلـ SQLiteـ.

- دعم مدمج للبحث النصي الكامل: يحتوي PostgreSQL على أدوات مثل `to_tsquery` و `to_tsvector` لتحويل النصوص إلى متوجهات بحث ومعالجة الاستعلامات.
 - الأداء العالي: يمكن إنشاء فهارس (indexes) لتسريع عمليات البحث، حتى مع قواعد بيانات كبيرة.
 - ميزات متقدمة: مثل التصنيف (ranking) وإعطاء أوزان (weights) للحقول المختلفة (مثل إعطاء الأولوية للعنوان على المحتوى).
 - تكامل مع Django: Django يوفر واجهة `django.contrib.postgres` لاستخدام هذه الميزات بسهولة.

مشروعك، تم استخدام `django.contrib.postgres.search.SearchVector` لإنشاء متوجه بحث يشمل حقول العنوان (`title`) والمحتمم (`body`) للمنشآت.

3. اعداد البحث في PostgreSQL مع Django

لتمكين البحث النصي الكامل في المشروع، تم اتخاذ الخطوات التالية:

أ. اضافة INSTALLED_APPS : django.contrib.postgres

في ملف `settings.py` ، تم اضافة 'django.contrib.postgres' إلى قائمة التطبيقات المنشطة:

هذا يتيح استخدام أدوات البحث النصي الكامل مثل `SearchQuery` و `SearchVector` في `Django`.

ب. إعداد قاعدة البيانات

تم إنشاء قاعدة بيانات PostgreSQL باستخدام Docker

```
bash ... ⌂ Copy  
docker run --name blog_db -e POSTGRES_DB=blog -e POSTGRES_USER=blog -e POSTGRES_PASSWORD
```

وتم تهيئة إعدادات قاعدة البيانات في `settings.py`

python

... Copy

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'blog',
        'USER': 'blog',
        'PASSWORD': '281133',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

ج. تحميل البيانات

تم تحميل البيانات إلى قاعدة البيانات باستخدام الأمر:

bash

... Copy

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

إذا واجهت مشكلة التكرار في `ContentType` (كما في السجلات السابقة)، يمكن حلها عن طريق:

- تعديل `.contenttypes.contenttype` لـ`mysite_data.json`
- أو إعادة إنشاء الملف مع استبعاد `:ContentType`

bash

... Copy

```
python manage.py dumpdata --exclude contenttypes --exclude auth.permission --indent=2 --output=mysite_data_cleaned.json
```

```
python manage.py dumpdata --exclude contenttypes --exclude auth.permission --indent=2 --output=mysite_data_cleaned.json
```

```
python manage.py loaddata mysite_data_cleaned.json
```

4. تفاصيل الكود المتعلق بالبحث

أ. نموذج البحث (SearchForm) في forms.py

تم إنشاء نموذج بسيط للبحث في ملف blog/forms.py

python

... Copy

```
from django import forms

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

- الوصف: هذا النموذج يحتوي على حقل واحد (query) يسمح للمستخدم بإدخال كلمات البحث.
- الغرض: يستخدم للنقط إدخال المستخدم وتمريره إلى العرض (view) لمعالجة البحث.
- البساطة: النموذج بسيط للغاية، لكنه كافٍ لتلبية احتياجات البحث الأساسية.

ب. عرض البحث (post_search) في views.py

العرض post_search هو المسؤول عن معالجة طلبات البحث وإرجاع النتائج. الكود:

python

... Copy

```
from django.contrib.postgres.search import SearchVector
from .forms import CommentForm, EmailPostForm, 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(
                query)
    return render(request, 'blog/post/search.html', {'form': form, 'query': query, 'resu
```

تحليل الكود خطوة بخطوة:

1. إنشاء النموذج:

- يتم إنشاء كائن `SearchForm` فارغ في البداية (`SearchForm()`). إذا كان هذا هو الطلب الأول (بدون استعلام)، يتم عرض النموذج الفارغ.

2. التحقق من وجود استعلام:

- يتم التحقق مما إذا كان هناك معلمة `query` في `request.GET` (لأن النموذج يُرسل باستخدام طريقة GET).
إذا وُجدت، يتم إنشاء النموذج باستخدام البيانات المُرسلة (`form = SearchForm(request.GET)`).

3. التحقق من صحة النموذج:

- إذا كان النموذج صالحًا (form.is_valid()), يتم استخراج قيمة الاستعلام من الحقل .form.cleaned_data['query']

٤. تنفيذ البحث:

- يتم البحث في المنشورات المنشورة فقط (Post.published).
 - يتم إنشاء متوجه بحث باستخدام `SearchVector('title', 'body')`، مما يعني أن البحث سيتم في حقول العنوان (`title`) والمحظى (`body`).
 - يتم تصفية المنشورات التي تتطابق مع الاستعلام باستخدام `.filter(search=query)`.
 - النتائج تخزن في `results`.

5. ارجاع النتائج:

- يتم تمرير النموذج (`form`), الاستعلام (`query`), والنتائج (`results`) إلى القالب `blog/post/search.html` لعرضها.

ملاحظات:

- استخدام `GET` بدلاً من `POST` يجعل رابط البحث (مثل `http://127.0.0.1:8000/blog/search/?`) قابلاً للمشاركة. `query=post` هو مدير مخصص (`custom manager`) يقتصر على المنشورات التي لها حالة `PUBLISHED`. `Post.published`

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.

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:

Posts containing "django"

Found 2 results

[Who was Django Reinhardt?](#)

Jean Reinhardt, known to all by his Romani nickname Django, was a ...

[Markdown post](#)

This is a post formatted with markdown

This is emphasized and this ...

[Search again](#)

1. ما هي الجذرية (Stemming)؟

الجذرية هي عملية تقليص الكلمات إلى شكلها الجذري أو الأساسي (stem). الهدف هو تحويل الكلمات المتشابهة (التي تختلف بسبب التصريفات أو الاشتراكات) إلى شكل موحد يسهل البحث عنه. على سبيل المثال:

- الكلمات "موسيقى"، "موسيقي"، و"موسيقية" قد تُعتبر متشابهة في محرك البحث.
- باستخدام الجذرية، يتم تحويل هذه الكلمات إلى الجذر "موسيقى" (music بالإنجليزية).

كيف تعمل الجذرية في البحث؟

- عندما يدخل المستخدم استعلام بحث (مثلاً "musical")، يقوم محرك البحث بتحويل الكلمة إلى جذرها ("music") باستخدام خوارزميات الجذرية.
- يتم بعد ذلك البحث عن الجذر في قاعدة البيانات، مما يسمح بمقابلة الكلمات المتشابهة (مثلاً "music" و "musicality").
- هذا يحسن دقة البحث ويضمن إرجاع نتائج ذات صلة حتى لو لم تكن الكلمة مطابقة تماماً.

الجذرية في PostgreSQL و Django

- الذي يقوم تلقائياً بتطبيق خوارزميات الجذرية على مصطلحات البحث `SearchQuery` يوفر كلان `Django` مما يضمن معالجة (مثل `to_tsquery` و `to_tsvector`) يدعم الجذرية من خلال أدوات البحث النصي الكامل `PostgreSQL`.
- الكلمات بشكل فعال.

2. إزالة الكلمات الشائعة (Stop Words)

الكلمات الشائعة (Stop Words) هي كلمات تُستخدم بشكل متكرر في اللغة ولا تحمل قيمة كبيرة في سياق البحث، مثل:

- بالإنجليزية: "a", "the", "on", "of".
- بالعربية: "في", "على", "و", "من".

لماذا تُزال الكلمات الشائعة؟

- هذه الكلمات تظهر كثيرًا في النصوص، مما يجعلها غير مفيدة لتحديد مدى ارتباط النتائج بالبحث.
- إزالتها تقلل من حجم البيانات التي يتم معالجتها وتسرّع عملية البحث.

كيف تُزال في PostgreSQL؟

- عند إنشاء استعلام بحث باستخدام `to_tsquery` (يتم ذلك داخليًا عبر `SearchQuery` في Django)، يقوم PostgreSQL تلقائيًا بتجاهل الكلمات الشائعة بناءً على قاموس اللغة المستخدم (مثل الإنجليزية).
- على سبيل المثال، إذا أدخل المستخدم "the music"، يتم تحويل الاستعلام إلى "music" فقط.

3. تصنیف النتائج (Ranking)

تصنيف النتائج هو عملية ترتيب نتائج البحث بناءً على مدى ارتباطها باستعلام البحث. PostgreSQL يوفر دالة تصنیف تُسمى `ts_rank` (يتم الوصول إليها في `SearchRank` عبر Django)، والتي تحسب درجة ارتباط كل نتائج بناءً على:

- عدد مرات ظهور مصطلحات الاستعلام: كلما زاد عدد المرات التي تظهر فيها كلمة البحث في النص، زادت درجة الارتباط.
- القرب بين المصطلحات: إذا كانت كلمات الاستعلام قريبة من بعضها في النص (مثلاً "Django framework" بدلاً من "Django ... framework")، يتم إعطاء درجة أعلى.
- الأوزان (إن وجدت): يمكن إعطاء أولوية لحقول معينة (مثل العنوان على المحتوى).

فائدۃ التصنیف:

- يضمن ظهور النتائج الأكثر صلة في أعلى القائمة.
- يحسن تجربة المستخدم من خلال تقديم المنشورات الأكثر ارتباطاً أولاً.

4. تعديل الكود لدعم الجذرية والتصنیف

تم تعديل عرض `post_search` في ملف `blog/views.py` لإضافة دعم الجذرية (باستخدام `SearchRank`) والتصنيف (باستخدام `SearchRank`). الكود المحدث مع الشرح:

أ. الاستيرادات الجديدة

تمت إضافة الاستيرادات التالية إلى `views.py`:

```
python
```

...

Copy

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

- `SearchVector` و `body` (مثل `title`) يستخدم لإنشاء متجه بحث يجمع النصوص من حقول متعددة.
- `SearchQuery`: يحول استعلام المستخدم إلى كائن استعلام بحث، مع تطبيق الجذرية وإزالة الكلمات الشائعة.
- `SearchRank`: يحسب درجة ارتباط كل نتائج باستعلام لترتيب النتائج.

1. إنشاء النموذج والمتغيرات:

- يُنشئ نموذج بحث فارغاً لعرضه في القالب : `form = SearchForm()`.
- يتم تهيئة متغيرات لتخزين الاستعلام ونتائج البحث : `query = None` و `results = []`.

2. التحقق من وجود استعلام:

- `if 'query' in request.GET` يتحقق مما إذا تم إرسال استعلام عبر طريقة `/blog/search/?query=django`.
- يُنشئ النموذج باستخدام البيانات المُرسلة : `form = SearchForm(request.GET)`.
- `if form.is_valid()` يتحقق من صحة البيانات، ثم يستخرج الاستعلام : `query = form.cleaned_data['query']`.

3. إنشاء متوجه البحث:

- يُنشئ متوجه بحث يجمع النصوص من حقلين : `search_vector = SearchVector('title', 'body')`. لكل منشور `title` و `body`.
- هذا المتوجه يحول النصوص إلى صيغة قابلة للبحث `(tsvector في PostgreSQL)`.

4. إنشاء استعلام البحث:

- إلى كائن استعلام بحث `SearchQuery(query: "django")` يحول الاستعلام المدخل.
- يقوم تلقائياً بما يلي `SearchQuery`:
 - الجذرية: تحويل الكلمات إلى جذورها (مثل "musical" إلى "music").
 - إزالة الكلمات الشائعة: تجاهل كلمات مثل "and" أو "the".
 - تحويل الاستعلام إلى صيغة `tsquery` (مثل `music` : * للبحث عن الكلمات التي تبدأ بـ "music").

5. تنفيذ البحث وتصنيف النتائج:

- يُضيف حقلين مؤقتين إلى النتائج : `results = Post.published.annotate(...)`.
- يخزن متوجه البحث لكل منشور : `search=search_vector`.
- يحسب درجة ارتباط كل منشور بالاستعلام بناءً على عدد المطابقات والقرب بين الكلمات.
- يقتصر على المنشورات التي تتطابق مع الاستعلام : `.filter(search=search_query)`.
- يرتيب النتائج تنازلياً حسب درجة الارتباط (الأعلى أولاً) : `.order_by('-rank')`.

6. إرجاع النتائج:

- يتم تمرير `blog/post/search.html` و `results` إلى القالب `form` و `query` لعرضها.

التغييرات الرئيسية مقارنة بالكود السابق:

- إضافة `SearchQuery`: يحل محل التصفية المباشرة بـ `filter(search=query)`، مما يتبع الجذرية وإزالة الكلمات الشائعة.
- إضافة `SearchRank`: يُضيف حقل `rank` لتصنيف النتائج حسب الارتباط.
- ترتيب النتائج بـ `order_by('-rank')`: يضمن ظهور النتائج الأكثر صلة أولاً.

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 stop words in Spanish:

```
def post_search(request):  
    #####  
    if form.is_valid():  
        query = form.cleaned_data['query']  
        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')  
        return render(request, 'blog/post/search.html', {'form': form, 'query': query, 'results': results})
```

1. ما هي كلمات الوقف (Stop Words)؟

كلمات الوقف هي كلمات شائعة في أي لغة لا تحمل معنى مميز بمفردها، مثل "في"، "على"، "من"، "هذا" بالعربية، أو "the", "is", "and" بالإنجليزية. هذه الكلمات تُستبعد عادةً أثناء معالجة النصوص (مثل البحث أو تحليل النصوص) لأنها لا تُضيف قيمة كبيرة لفهم المحتوى وتُقلل من ضوضاء البيانات.

2. ما هو التصريف اللغوي (Stemming)؟

التصريف اللغوي هو عملية تقليل الكلمات إلى جذورها الأساسية. على سبيل المثال، بالعربية، كلمات مثل "كتب"، "كتاب"، "يكتب" قد تختصر إلى الجذر "كتب". هذا يساعد في مطابقة المتغيرات المختلفة لنفس الكلمة أثناء البحث.

3. كيف يعمل الكود المذكور؟

الكود يُظهر دالة `post_search` التي تعالج طلب بحث في مدونة. الهدف هو البحث في عناوين ومحطيات المقالات (Posts) بناءً على استعلام المستخدم. الكود يستخدم مكتبة PostgreSQL Full-Text Search في Django، وهي تدعم معالجة النصوص بلغات مختلفة.

الخطوات الرئيسية في الكود:

- إنشاء نموذج البحث (SearchForm): يُستخدم لأخذ استعلام المستخدم (query).
- التحقق من صحة الاستعلام: إذا كان الاستعلام صالحًا، يتم استخدامه للبحث.
- إعداد `SearchVector`: يُحدد الحقول التي سيتم البحث فيها (هنا: `title` و `body`)، مع تحديد اللغة باستخدام `config='spanish'`. هذا يعني أن المعالجة اللغوية (مثل التصريف وإزالة كلمات الوقف) ستتم وفقًا لقواعد اللغة الإسبانية.
- إعداد `SearchQuery`: يأخذ استعلام المستخدم ويطبق عليه نفس إعدادات اللغة (`config='spanish'`).
- تنفيذ البحث: يتم البحث في المقالات باستخدام `SearchQuery` و `SearchVector`، مع ترتيب النتائج حسب الصلة `.SearchRank` (Rank) باستخدام.

: 'config='spanish دور

- عند تحديد `config='spanish'`، يستخدم PostgreSQL قاموساً وقواعد خاصة باللغة الإسبانية.
 - يقوم النظام تلقائياً بإزالة كلمات الوقف الإسبانية (مثل "el", "la", "de").
 - يُطبق التصريف اللغوي للكلمات الإسبانية (مثل تحويل "corriendo" إلى "corr").

4. كيف يمكن تطبيق ذلك على اللغة العربية؟

للتغيير نفس العملية باللغة العربية، يمكن تغيير `config='arabic'` إلى `config='spanish'`، بشرط أن تكون قاعدة البيانات (PostgreSQL) مهأة لدعم اللغة العربية. الخطوات ستكون كالتالي:

تعديل الكود للغة العربية:

```
python ... ⌂ Copy

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', config='arabic')
            search_query = SearchQuery(query, config='arabic')
            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})
```

ما يحدث عند استخدام : 'config='arabic'

- إزالة كلمات الوقف العربية: سيتم استبعاد كلمات مثل "في"، "على"، "من"، "هذا"، وغيرها من كلمات الوقف الشائعة في اللغة العربية.
- التصريف اللغوي: سيتم تحويل الكلمات إلى جذورها (على سبيل المثال، "كتابة"، "كتب"، "يكتب" قد تختصر إلى "كتب").
- تحسين البحث: بفضل إزالة كلمات الوقف والتصريف، يصبح البحث أكثر دقة، حيث يركز على الكلمات ذات المعنى فقط.

متطلبات إضافية للغة العربية:

1. دعم PostgreSQL للغة العربية:

- يجب تثبيت إعدادات اللغة العربية في PostgreSQL. يمكن التحقق من ذلك باستخدام الأمر:

```
sql ... Copy
SELECT * FROM pg_ts_config WHERE cfgname = 'arabic';
```

- إذا لم يكن موجوداً، قد تحتاج إلى إضافة قاموس عربي مخصص.

2. إعداد قاموس كلمات الوقف: PostgreSQL يحتوي على قوائم افتراضية لكلمات الوقف للغات مختلفة، بما في ذلك العربية. يمكن تخصيص هذه القائمة إذا لزم الأمر.

3. معالجة تعقيبات اللغة العربية: اللغة العربية تحتوي على تحديات مثل الحركات والتشكيل، لذا قد تحتاج إلى أدوات إضافية مثل مكتبة `snowball` للتصريف اللغوي.

5. مثال عملي بالعربية:

افترض أن المستخدم أدخل استعلاماً: "في الكتابة عن الكتب".

- بدون إزالة كلمات الوقف: سيتم البحث عن كل كلمة، بما في ذلك "في" و"عن"، مما قد يؤدي إلى نتائج غير دقيقة.
- مع إزالة كلمات الوقف: سيتم استبعاد "في" و"عن"، وسيتم التركيز على "الكتابة" و"الكتب".
- مع التصريف: سيتم تحويل "الكتابة" و"الكتب" إلى الجذر "كتب"، مما يعني أن البحث سيطابق أي مقالة تحتوي على أي صيغة من الكلمة (مثل "كتاب"، "يكتب").

Weighting queries

We can boost specific vectors so that more weight is attributed to them when ordering results by relevance. 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.

ما الجديد في الكود؟

الجديد هو إضافة الأوزان (Weights) إلى البحث. الأوزان تحدد أهمية جزء معين من المقالة (مثل العنوان أو المحتوى) عند ترتيب النتائج، بمعنى آخر، يمكننا جعل العنوان "أهم" من المحتوى، بحيث إذا وجدت الكلمة المبحوث عنها في العنوان، تظهر المقالة في أعلى النتائج.

شرح الكود خطوة بخطوة بطريقة بسيطة:

1. نموذج البحث (SearchForm):

- عندما يدخل المستخدم كلمة بحث (مثل "كتاب")، يتم إرسالها عبر نموذج يسمى `SearchForm`.
- الكود يتحقق إذا كانت الكلمة صالحة (غير فارغة أو غير صحيحة).

2. إعداد متغيرات البحث (SearchVector):

- الكود يحدد أين سيتم البحث: في العنوان (`title`) والمحتوى (`body`).
- الجديد هنا هو إضافة الأوزان:

- العنوان يأخذ وزن `A` (وهو يساوي 1.0، أعلى وزن).
- المحتوى يأخذ وزن `B` (وهو يساوي 0.4، وزن أقل).

هذا يعني أن الكلمات الموجودة في العنوان ستكون أكثر أهمية من الكلمات في المحتوى.

3. إعداد استعلام البحث (SearchQuery):

- يأخذ الكلمة التي أدخلها المستخدم (مثل "كتاب") ويستعد للبحث عنها.

4. تنفيذ البحث وترتيب النتائج:

- النظام يبحث في المقالات (Posts) ويحسب درجة الأهمية (Rank) لكل مقالة بناءً على:
 - هل الكلمة موجودة في العنوان أو المحتوى؟
 - ما هو وزن الحقل (العنوان أو المحتوى)؟
- المقالات التي تحتوي على الكلمة في العنوان ستحصل على درجة أعلى (لأن وزن العنوان 1.0) مقارنة بالمقالات التي تحتوي على الكلمة في المحتوى فقط (وزن 0.4).
 - يتم استبعاد أي مقالة لها درجة أقل من 0.3 (لضمان عرض النتائج ذات الصلة فقط).
 - النتائج تُرتّب من الأعلى إلى الأدنى حسب درجة الأهمية (`order_by(' -rank)`).

5. إرجاع النتائج:

- يتم إرسال النتائج إلى صفحة ويب (search.html) تُظهر نموذج البحث، الكلمة المبحوث عنها، والمقالات المطابقة.

ما هي الأوزان (Weights) وكيف تعمل؟

- الأوزان هي أرقام تحدد أهمية جزء من النص (مثل العنوان أو المحتوى).
- في PostgreSQL (قاعدة البيانات المستخدمة هنا)، هناك 4 أوزان افتراضية:

- $A = 1.0$ (الأعلى أهمية)
- $B = 0.4$
- $C = 0.2$
- $D = 0.1$ (الأقل أهمية)

• في الكود:

- العنوان يأخذ وزن (1.0) A ، لذا إذا وجدت الكلمة البحث في العنوان، ستكون المقالة في صدارة النتائج.
- المحتوى يأخذ وزن (0.4) B ، لذا إذا وجدت الكلمة في المحتوى فقط، ستكون أقل أهمية.

مثال عملي لتوضيح الفكرة:

افتراض أنك تبحث عن الكلمة "كتاب"، وهناك مقالتان:

- المقالة 1: العنوان: "أفضل كتاب" ، المحتوى: "معلومات عامة".
- المقالة 2: العنوان: "معلومات عامة" ، المحتوى: "كتاب رائع".

بدون أوزان:

- النظام قد يعطي كلا المقالتين درجات متساوية تقريباً، لأنه لا يفرق بين العنوان والمحتوى.

مع الأوزان (كما في الكود):

- المقالة 1 ستحصل على درجة أعلى لأن الكلمة "كتاب" في العنوان (وزن 1.0).
- المقالة 2 ستحصل على درجة أقل لأن الكلمة "كتاب" في المحتوى (وزن 0.4).
- النتيجة: المقالة 1 ستظهر في الأعلى، حتى لو كانت الكلمة موجودة في كلا المقالتين.

تأثيرات إضافة الأوزان:

1. تحسين تجربة المستخدم:

- النتائج تصبح أكثر دقة، لأن العنوان غالباً تكون أكثر تعبيراً عن موضوع المقالة.
- المستخدم يجد المقالات ذات الصلة بسرعة أكبر.

2. ترتيب أفضل للنتائج:

- المقالات التي تحتوي على الكلمة في العنوان تظهر أولاً، مما يعكس غالباً أهميتها.

3. تصفية النتائج غير المهمة:

- شرط `rank_gte=0.3` يضمن استبعاد المقالات التي ليست ذات صلة قوية (مثل تلك التي تحتوي على الكلمة بشكل عابر).

4. مرونة أكبر:

- يمكنك تغيير الأوزان حسب احتياجاتك. على سبيل المثال، إذا أردت إعطاء المحتوى أهمية أكبر، يمكنك جعل وزنه `B` والعنوان `A`.

لماذا هذا مهم؟

بدون الأوزان، قد تظهر مقالة تحتوي على الكلمة في المحتوى بشكل متكرر (لكن ليست ذات صلة) قبل مقالة تحتوي على الكلمة في العنوان (وهي عادةً أكثر صلة). الأوزان تُعطي النظام "ذكاءً" لتحديد ما هو أكثر أهمية للمستخدم.

كيف يمكنني تجربة هذا بنفسي؟

إذا كنت تستخدم PostgreSQL و Django:

1. تأكد من تثبيت مكتبة `django.contrib.postgres` لدعم البحث النصي الكامل.
2. أضف الكود أعلاه إلى ملف `views.py` في تطبيقك.
3. جرب البحث بكلمات مختلفة ولاحظ كيف تظهر المقالات التي تحتوي على الكلمة في العنوان أولاً.

الخلاصة بجملة واحدة:

الكود يجعل البحث أذكى بإعطاء العنوان أهمية أكبر (وزن 1.0) من المحتوى (وزن 0.4)، فيظهر المقالات التي تحتوي على الكلمة البحث في العنوان أولاً مع استبعاد النتائج غير المهمة (درجة أقل من 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` database extension first. Django provides database migration operations to create PostgreSQL extensions. Let's add a migration that creates the extension in the database.

First, execute the following command in the shell prompt to create an empty migration:

```
python manage.py makemigrations --name=trigram_ext --empty blog
```

This will create an empty migration for the blog application. You will see the following output:

```
Migrations for 'blog':  
  blog/migrations/0005_trigram_ext.py
```

Edit the file blog/migrations/0005_trigram_ext.py and add the following lines highlighted in bold:

```
from django.contrib.postgres.operations import TrigramExtension  
  
from django.db import migrations  
  
class Migration(migrations.Migration):  
  
    dependencies = [  
  
        ('blog', '0004_post_tags'),  
  
    ]  
  
    operations = [  
  
        TrigramExtension(),  
  
    ]
```

You have added the TrigramExtension operation to the database migration. This operation executes the SQL statement CREATE EXTENSION pg_trgm to create the extension in PostgreSQL.

Now execute the migration with the following command:

```
python manage.py migrate blog
```

You will see the following output:

```
Running migrations:  
  Applying blog.0005_trigram_ext... OK
```

The pg_trgm extension has been created in the database. Let's modify post_search 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']  
            search_vector = SearchVector('title', weight='A')+  
SearchVector('body', weight='B')  
            search_query = SearchQuery(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:

Posts containing "yango"

Found 1 result

Who was Django Reinhardt?

Jean Reinhardt, known to all by his Romani nickname Django, was a ...

[Search again](#)

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

ما هي الثلاثيات (Trigrams)؟

الثلاثيات هي مجموعة من ثلاثة أحرف متتالية في كلمة أو نص. على سبيل المثال، إذا كانت الكلمة هي "Django"، فإن الثلاثيات هي:

- Dja
- jan
- ang
- ngo

عندما نقارن بين نصين (مثل كلمة البحث والعنوان في قاعدة البيانات)، نحسب عدد الثلاثيات المشتركة بينهما. كلما زاد عدد الثلاثيات المشتركة، كلما اعتبر النصان أكثر تشابهًا. هذه الطريقة مفيدة جدًا للبحث حتى لو كان هناك أخطاء إملائية (مثل كتابة "yango" بدلاً من "Django").

ما الذي فعلناه في الكود؟

لنستعرض الخطوات التي تم تنفيذها في الكود لإضافة خاصية البحث باستخدام الثلاثيات في Django:

1. تثبيت إضافة pg_trgm في PostgreSQL

يمكنك مساعد على حساب تشابه الثلاثيات `pg_trgm` لديها إضافة تسمى (قاعدة البيانات التي تستخدمها) باستخدام الأمر (Migration) (Migration) لتفعيل هذه الإضافة، فهذا يإنشاء ملف هجرة:

```
text
...
Copy
```

```
python manage.py makemigrations --name=trigram_ext --empty blog
```

هذا الأمر أنشأ ملف هجرة فارغاً.

- لإضافة الأمر (0005_trigram_ext.py) تم فتحه وتعديل ملف الهجرة:

```
python
...
Copy
```

```
TrigramExtension()
```

التي تدعم حساب الثلاثيات PostgreSQL `pg_trgm` يفضل إضافة هذا الأمر يختار.

- بعد ذلك، نفذ المиграة باستخدام:

```
text
...
Copy
python manage.py migrate blog
```

هذا الأمر قام بتحديث الإخفاقة في قاعدة البيانات.

2. تعديل طريقة البحث في Django

- في ملف `views.py`، أضفنا استيراد الأداة التي ستساعدنا على استخدام الثلاثيات:

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

- ثم قمنا بتعديل دالة البحث `post_search` لتسخدم `TrigramSimilarity` بدلًا من البحث العادي. الدالة الجديدة تبحث في عنوان المقالات (`title`) وتقارن كلمة البحث (`query`) مع العنوانين باستخدام الثلاثيات.
- ال코드 المهم في الدالة هو:

```
python
...
Copy
TrigramSimilarity('title', query)).filter(similarity__gt=0.1).order_by('-similarity')
```

دعنا نفكك هذا السطر:

- مع حقل (`query`) أن يقارن كلمة البحث Django هنا نقول له: `TrigramSimilarity('title', query)`.
- باستخدام الثلاثيات، ويعطي كل نتيجة درجة تشابه (من 0 إلى 1) (عنوان `title`)
- نضيف درجة التشابه كحقل مؤقت لكل مقالة: `annotate(similarity=...)`.
- نستبعد النتائج التي تكون درجة تشابهها أقل من 0.1 (أي النتائج غير `filter(similarity__gt=0.1)`).
- نرتتب النتائج بحيث تظهر النتائج الأكثر تشابهًا أولاً: `order_by('-similarity')`.