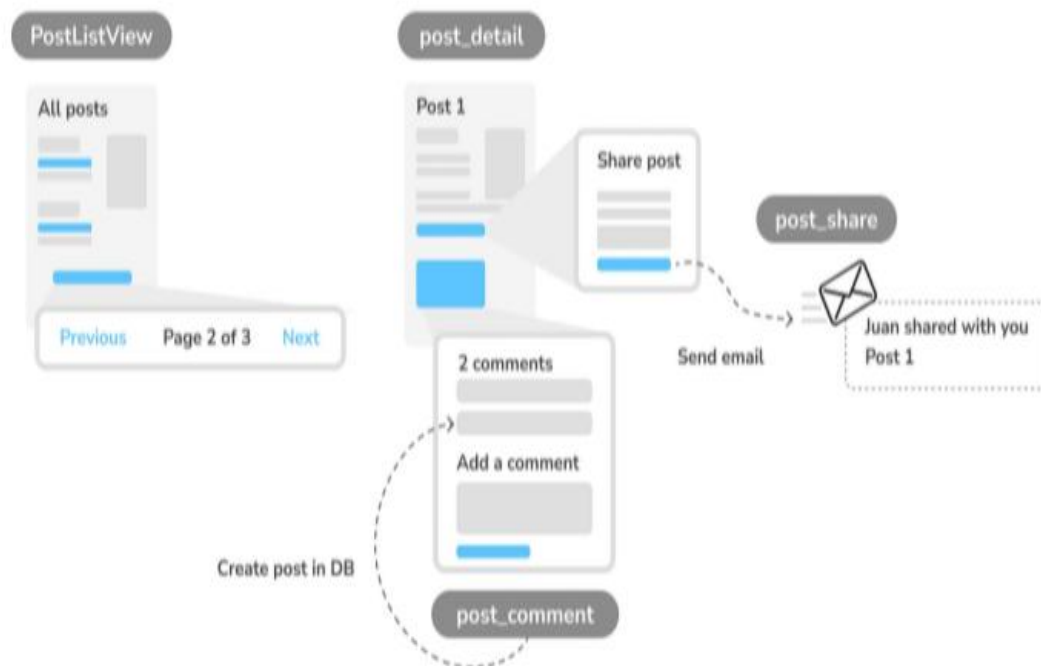


CH2: Enhancing Your Blog and Adding Social Features

Functional overview



The source code for this chapter can be found at <https://github.com/PacktPublishing/Django-5-by-example/tree/main/Chapter02>.

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

Using canonical URLs for models

A canonical URL is the preferred URL for a resource. You can think of it as the URL of the most representative page for specific content.

Canonical URLs allow you to specify the URL for the master copy of a page. Django allows you to implement the `get_absolute_url()` method in your models to return the canonical URL for the object.

Edit the `models.py` file of the blog application to import the `reverse()` function and add the **`get_absolute_url()`** method to the `Post` model as follows. The new code is highlighted in bold:

```
from django.urls import reverse
```

```
#####
```

```
class Post(models.Model):
```

```
#####
```

```
    def get_absolute_url(self):  
        return reverse( 'blog:post_detail', args=[self.id] )
```

The reverse() function will build the URL dynamically using the URL name defined in the URL patterns.

This URL has a required parameter, which is the id of the blog post to retrieve. We have included the id of the Post object as a positional argument by using **args=[self.id]**

Edit the blog/post/list.html file and replace the following line:

```
<a href="{% url 'blog:post_detail' post.id %}">
```

Replace the preceding line with the following line:

```
<a href="{% post.get_absolute_url %}">
```

Open the shell prompt and execute the following command to start the development server: **python manage.py runserver**

Creating SEO-friendly URLs for posts

The canonical URL for a blog post detail view currently looks like /blog/1/. We will change the URL pattern to create SEO-friendly URLs for posts. We will be using both the publish date and slug values to build the URLs for single posts. By combining dates, we will make a post detail URL to look like /blog/2024/1/1/who-was-django-reinhardt/. We will provide search engines with friendly URLs to index.

To retrieve single posts with the combination of publication date and slug, we need to ensure that no post can be stored in the database with the same slug and publish date as an existing post. We will prevent the

Post model from storing duplicated posts by defining slugs to be unique for the publication date of the post.

Edit the models.py file and add the following unique_for_date parameter to the slug field of the Post model:

```
class Post(models.Model):
```

```
#####
```

```
slug = models.SlugField( max_length=250,unique_for_date='publish' )
```

```
#####
```

By using unique_for_date, the slug field is now required to be unique for the date stored in the publish field. Note that the publish field is an instance of DateTimeField, but the check for unique values will be done only against the date (not the time). Django will prevent you from saving a new post with the same slug as an existing post for a given publication date. We have now ensured that slugs are unique for the publication date, so we can now retrieve single posts by the publish and slug fields.

We have changed our models, so, let's create migrations. Note that unique_for_date is not enforced at the database level, so no database migration is required. However, Django uses migrations to keep track of all model changes. We will create a migration just to keep migrations aligned with the current state of the model.

Run the following command in the shell prompt:

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

Execute the following command in the shell prompt to apply existing migrations:

```
python manage.py migrate
```

No action will be done in the database because unique_for_date is not enforced at the database level.

Modifying the URL patterns

Let's modify the URL patterns to use the publication date and slug for the post detail URL.

Edit the urls.py file of the blog application and replace the following line:

```
path('<int:id>/', views.post_detail, name='post_detail'),
```

Replace the preceding line with the following lines:

```
path(,('int:year>/<int:month>/<int:day>/<slug:post>/', views.post_detail, name='post_detail>'
```

The URL pattern for the post_detail view takes the following arguments:

- year: This requires an integer
- month: This requires an integer
- day: This requires an integer
- ,post: This requires a slug (a string that contains only letters .(numbers, underscores, or hyphens

The int path converter is used for the year, month, and day parameters, whereas the slug path con-verter is used for the post parameter.

Modifying the views

We will change the parameters of the post_detail view to match the new URL parameters and use

them to retrieve the corresponding Post object.

Edit the views.py file and edit the post_detail view like this:

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

We have modified the post_detail view to take the year, month, day, and post arguments and retrieve a published post with the given slug and publication date. By adding unique_for_date='publish' to the slug

field of the Post model, we ensured that there would be only one post with a slug for a given date. Thus, you can retrieve single posts using the date and slug.

Modifying the canonical URL for posts

We also have to modify the parameters of the canonical URL for blog posts to match the new URL parameters.

Edit the models.py file of the blog application and edit the `get_absolute_url()` method as follows:

```
class Post(models.Model):  
  
    #####  
    def get_absolute_url(self)  
  
        return reverse( 'blog:post_detail', args=[ self.publish.year'  
( [ self.publish.month, self.publish.day, self.slug
```

Start the development server by typing the following command in the shell prompt: `python manage.py runserver`

The `get_absolute_url()` function is a method typically used in models to provide an absolute URL that points to a specific page associated with a specific instance of the model. This method is particularly useful when you need to return a dynamic URL to display details of the instance in your application.

Adding pagination

Instead of displaying all the posts on a single page, you may want to split the list of posts across several pages and include navigation links to the different pages. This functionality is called pagination, and you can find it in almost every web application that displays long lists of items.

Django has a built-in pagination class that allows you to manage paginated data easily. You can define the number of objects you want to be returned per page and you can retrieve the posts that correspond to the page requested by the user.

Adding pagination to the post list view

We will add pagination to the list of posts so that users can easily navigate through all posts published on the blog.

Edit the views.py file of the blog application to import the Django Paginator class and modify the post_list view as follows:

```
from django.core.paginator import Paginator

from django.shortcuts import get_object_or_404, render

from .models import Post

def post_list(request):

    post_list = Post.published.all

    # Pagination with 3 posts per page

    paginator = Paginator(post_list, 3)

    page_number = request.GET.get('page', 1)

    posts = paginator.page(page_number)

    return render( request, 'blog/post/list.html', {'posts': posts} )
```

Let's review the new code we have added to the view:

1. We instantiate the Paginator class with the number of objects to return per page. We will display three posts per page.
2. We retrieve the page GET HTTP parameter and store it in the page_number variable. This parameter contains the requested page number. If the page parameter is not in the GET parameters of the request, we use the default value 1 to load the first page of results.
3. We obtain the objects for the desired page by calling the page() method of Paginator. This method returns a Page object that we store in the posts variable.
4. We pass the posts object to the template.

Creating a pagination template

We need to create a page navigation for users to browse through the different pages. In this section, we will create a template to display the pagination links, and we'll make it generic so that we can reuse the template for any object pagination on our website.

In the templates/ directory, create a new file and name it pagination.html. Add the following HTML code to the file:

```
<"div class="pagination>

<"span class="step-links>

{% if page.has_previous %}

<a href="?page={{ page.previous_page_number }}">Previous</a>

{% endif %}

<"span class="current>

.Page {{ page.number }} of {{ page.paginator.num_pages }}

<span/>

{% if page.has_next %}

<a href="?page={{ page.next_page_number }}">Next</a>

{% endif %}

<span/>

<div/>
```

This is the generic pagination template. The template expects to have a Page object in the context to render the previous and next links and to display the current page and total pages of results.

Let's return to the blog/post/list.html template and include the pagination.html template at the bottom of the {% content %} block, as follows:

```

{% "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="date">

Published {{ post.publish }} by {{ post.author }}

<p/>

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

{% endfor %}

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

{% endblock %}

```

The `{% include %}` template tag loads the given template and renders it using the current template context. We use `with` to pass additional context variables to the template. The pagination template uses the `page` variable to render, while the `Page` object that we pass from our view to the template is called `posts`. We use `with page=posts` to pass the variable expected by the pagination template.

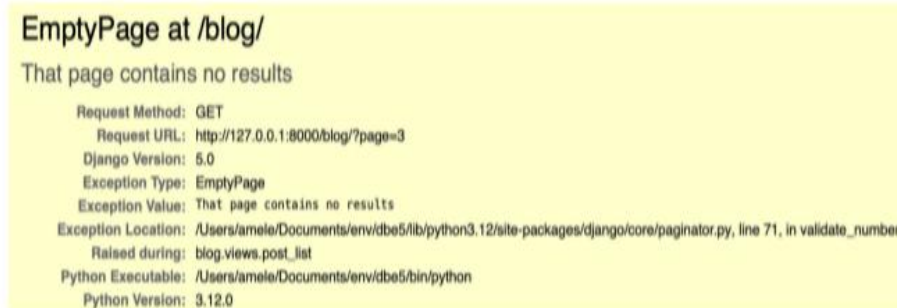
Start the development server by typing the following command in the shell prompt: `python manage.py runserver`

If you click on Next, you will see the last post. The URL for the second page contains the `?page=2` GET parameter. This parameter is used by the view to load the requested page of results using the paginator.

Handling pagination errors

The page parameter used by the view to retrieve the given page could potentially be used with wrong values, such as non-existing page numbers or a string value that cannot be used as a page number. We will implement appropriate error handling for those cases.

Open `http://127.0.0.1:8000/blog/?page=3` in your browser. You should see the following error page:



The Paginator object throws an EmptyPage exception when retrieving page 3 because it's out of range.

There are no results to display. Let's handle this error in our view.

Edit the `views.py` file of the blog application to add the necessary imports and modify the `post_list` view as follows

```
from django.core.paginator import EmptyPage, Paginator
```

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

```
from .models import Post
```

```
def post_list(request):
```

```
    post_list = Post.published.all
```

```
    # Pagination with 3 posts per page
```

```
    paginator = Paginator(post_list, 3)
```

```
    page_number = request.GET.get('page', 1)
```

```
    try
```

```
        posts = paginator.page(page_number)
```

```
    except EmptyPage
```

```
        # If page_number is out of range get last page of results
```

```
        posts = paginator.page(paginator.num_pages)
```

```
    return render(request, 'blog/post/list.html', {'posts': posts})
```

We have added a try and except block to manage the EmptyPage exception when retrieving a page. If the page requested is out of range, we return the last page of results. We get the total number of pages with paginator.num_pages. The total number of pages is the same as the last page number.

Open <http://127.0.0.1:8000/blog/?page=asdf> in your browser. You should see the following error page:



In this case, the Paginator object throws a PageNotAnInteger exception when retrieving the page asdf because page numbers can only be integers. Let's handle this error in our view.

Edit the views.py file of the blog application to add the necessary imports and modify the post_list view as follows

```
from django.core.paginator import EmptyPage, PageNotAnInteger, Paginator
```

```
#####
```

```
#####
```

```
try:
```

```
posts = paginator.page(page_number)
```

```
:except PageNotAnInteger
```

```
If page_number is not an integer get the first page #
```

```
posts = paginator.page(1)
```

```
:except EmptyPage
```

```
If page_number is out of range get last page of results #
```

```
posts = paginator.page(paginator.num_pages)
return render( request, 'blog/post/list.html', {'posts': posts} )
```

We have added a new except block to manage the PageNotAnInteger exception when retrieving a page. If the page requested is not an integer, we return the first page of results.

class-based views

Class-based views are an alternative way to implement views as Python objects instead of functions.

Since a view is a function that takes a web request and returns a web response, Django provides base view classes that you can use to implement your own views. All of them inherit from the View class.

Why use class-based views

Class-based views offer some advantages over function-based views that are useful for specific use cases. Class-based views allow you to:

- ,Organize code related to HTTP methods, such as GET, POST or PUT, in separate methods, instead of using conditional branching
- Use multiple inheritance to create reusable view classes (also known as mixins).

Using a class-based view to list posts.

To understand how to write class-based views, we will create a new class-based view that is equivalent to the post_list view. We will create a class that will inherit from the generic ListView view offered by Django. ListView allows you to list any type of object.

Edit the views.py file of the blog application and add the following code to it:

```
from django.views.generic import ListView
```

```
class PostListView(ListView):
```

```
.....
```

```
Alternative post list view
```

```
.....
```

```
()queryset = Post.published.all
```

```
'context_object_name' = 'posts
```

```
paginate_by = 3
```

```
'template_name' = 'blog/post/list.html
```

We have defined a view with the following attributes:

- We use queryset to use a custom QuerySet instead of retrieving all objects. Instead of defining a queryset attribute we could have specified model = Post and Django would have built the generic Post.objects.all() QuerySet for us.
- We use the context variable posts for the query results. The **default variable** is **object_list** if you don't specify any context_object_name.
- We define the pagination of results with paginate_by, returning three objects per page.
- We use a custom template to render the page with template_name. If you don't set a default template, ListView will use blog/post_list.html by default.

Edit the urls.py file of the blog application, comment the preceding post_list URL pattern, and add a new URL pattern using the PostListView class, as follows:

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

In order to keep pagination working, we have to use the right page object that is passed to the template. Django's ListView generic view passes the page requested in a variable called page_obj. We have to **edit**

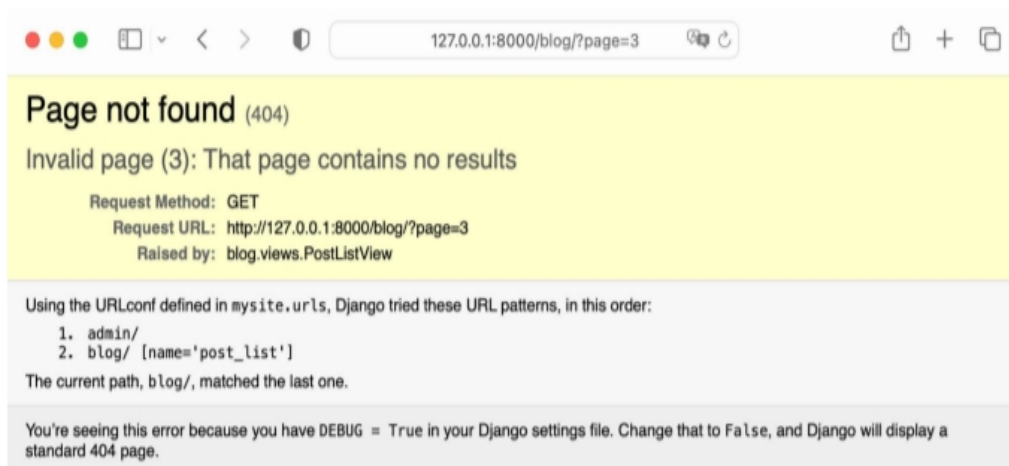
the post/list.html template accordingly to include the paginator using the right variable, as follows:

```
#####
```

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

```
{% endblock %}
```

The exception handling in this case is a bit different. If you try to load a page out of range or pass a non-integer value in the page parameter, the view will return an HTTP response with the status code 404 (page not found) like this:



The exception handling that returns the HTTP 404 status code is provided by the ListView view. This is a simple example of how to write class-based views.

Recommending posts by email

We will allow users to share blog posts with others by sending post recommendations via email.

To allow users to share posts via email, we will need to:

1. ,Create a form for users to fill in their name, their email address the recipient's email address, and optional comments
2. Create a view in the views.py file that handles the posted data and sends the email

3. Add a URL pattern for the new view in the urls.py file of the blog application
4. Create a template to display the form

Creating forms with Django

Let's start by building the form to share posts. Django has a built-in forms framework that allows you to create forms easily. The forms framework makes it simple to define the fields of the form, specify how they have to be displayed, and indicate how they have to validate input data. The Django forms framework offers a flexible way to render forms in HTML and handle data.

Django comes with two base classes to build forms:

- **Form**: This allows you to build standard forms by defining fields and validations.
- **ModelForm**: This allows you to build forms tied to model instances. It provides all the functionalities of the base Form class, but form fields can be explicitly declared, or automatically generated, from model fields. The form can be used to create or .edit model instances

First, create a **forms.py** file **inside** the directory of your blog application and add the following code to it:

```
from django import forms
```

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

```
    name = forms.CharField(max_length=25)
```

```
    email = forms.EmailField
```

```
    to = forms.EmailField
```

```
    comments = forms.CharField( required=False, widget=forms.Textarea )
```

We have defined our first Django form. The EmailPostForm form inherits from the base Form class. We use different field types to validate data accordingly.

Notice:

Forms can reside anywhere in your Django project. The convention is to place them inside a forms.py file for each application.

The form contains the following fields:

- **name:** An instance of CharField with a maximum length of 25 characters. We will use it for the name of the person sending the post.
- **email:** An instance of EmailField. We will use the email of the person sending the post recommendation
- **to:** An instance of EmailField. We will use the email address of the recipient, who will receive an email recommending the post
- **comments:** An instance of CharField. We will use it for comments to include in the post recommendation email. We have made this field optional by setting required to False, and we have specified a custom widget to render the field

Each field type has a default widget that determines how the field is rendered in HTML. The name field is an instance of CharField. This type of field is rendered as an `<input type="text">` HTML element. The default widget can be overridden with the widget attribute. In the comments field, we use the Textarea widget to display it as a `<textarea>` HTML element instead of the default `<input>` element.

Field validation also depends on the field type. For example, the email and to fields are EmailField fields. Both fields require a valid email address; the field validation will otherwise raise a forms.ValidationError exception and the form will not validate. Other parameters are also taken into account for the form field validation, such as the name field having a maximum length of 25 or the comments field being optional.

Handling forms in views

We have defined the form to recommend posts via email. Now, we need a view to create an instance of the form and handle the form submission.

Edit the views.py file of the blog application and add the following code to it:

```

from .forms import EmailPostForm

def post_share(request, post_id):

    Retrieve post by id #
    post = get_object_or_404( Post, id=post_id, status=Post.Status.PUBLISHED )

    :if request.method == 'POST'

    Form was submitted #

    form = EmailPostForm(request.POST)

    :()if form.is_valid

    Form fields passed validation #

    cd = form.cleaned_data

    send email ... #

    :else

    ()form = EmailPostForm

    return render( request, 'blog/post/share.html', {'post': post, 'form': form } )

```

We have defined the `post_share` view that takes the `request` object and the `post_id` variable as parameters. We use the `get_object_or_404()` shortcut to retrieve a published post by its id.

We use the same view both for displaying the initial form and processing the submitted data. The HTTP request method allows us to differentiate whether the form is being submitted. A GET request will indicate that an empty form has to be displayed to the user and a POST request will indicate the form is being submitted. We use `request.method == 'POST'` to differentiate between the two scenarios.

This is the process to display the form and handle the form submission:

1. When the page is loaded for the first time, the view receives a GET request. In this case, a new `EmailPostForm` instance is created and stored in the `form` variable. This form instance will be used to display the empty form in the template:

`form = EmailPostForm()`

2. When the user fills in the form and submits it via POST, a form instance is created using the submitted data contained in `request.POST`:

if request.method == 'POST':

Form was submitted #

form = EmailPostForm(request.POST)

3. After this, the data submitted is validated using the form's `is_valid()` method. This method validates the data introduced in the form and returns `True` if all fields contain valid data. If any field contains invalid data, then `is_valid()` returns `False`. The list of validation errors can be obtained with `form.errors`.

4. If the form is not valid, the form is rendered in the template again, including the data submitted. Validation errors will be displayed in the template.

5. If the form is valid, the validated data is retrieved with `form.cleaned_data`. This attribute is a dictionary of form fields and their values. Forms not only validate the data but also clean the data by normalizing it to a consistent format.

Notice:

If your form data does not validate, `cleaned_data` will contain only the valid fields.

Sending emails with Django

Sending emails with Django is very straightforward. You need to have a local SMTP server, or you need to access an external SMTP server, like your email service provider.

The following settings allow you to define the SMTP configuration to send emails with Django:

- **EMAIL_HOST**: The SMTP server host; the default is localhost
- **EMAIL_PORT**: The SMTP port; the default is 25
- **EMAIL_HOST_USER**: The username for the SMTP server
- **EMAIL_HOST_PASSWORD**: The password for the SMTP server
- **EMAIL_USE_TLS**: Whether to use a Transport Layer Security (TLS) secure connection
- **EMAIL_USE_SSL**: Whether to use an implicit TLS secure connection

Additionally, you can use the `DEFAULT_FROM_EMAIL` setting to specify the default sender when sending emails with Django. For this example, we will use Google's SMTP server with a standard Gmail account.

Working with environment variables

We will add SMTP configuration settings to the project, and we will load the SMTP credentials from environment variables. By using environment variables, we will avoid embedding credentials in the source code. There are multiple reasons to keep configuration separate from the code:

- **Security:** Credentials or secret keys in the code can lead to unintentional exposure, especially if you push the code to public repositories.
- **Flexibility:** Keeping the configuration separate will allow you to use the same code base across different environments without any changes.
- **Maintainability:** Changing a configuration won't require a code change ensuring that your project remains consistent across versions.

To facilitate the separation of configuration from code, we are going to use python-decouple. This library simplifies the use of environment variables in your projects.

First, install python-decouple via pip by running the following command:

```
python -m pip install python-decouple==3.8
```

Then, **create a new file inside your project's root directory and name it .env.**

The .env file will contain key-value pairs of environment variables. Add the following lines to the new file:

```
EMAIL_HOST_USER=your_account@gmail.com
```

```
EMAIL_HOST_PASSWORD=
```

```
DEFAULT_FROM_EMAIL=My Blog <your_account@gmail.com>
```

If you have a Gmail account, replace your_account@gmail.com with your Gmail account. The EMAIL_HOST_PASSWORD variable has no value yet, we will add it later. The DEFAULT_FROM_EMAIL variable will be used to specify the default sender for our emails. If you don't have a Gmail account, you can use the SMTP credentials for your email service provider.

If you are using a git repository for your code, make sure to include .env in the .gitignore file of your repository. By doing so, you ensure that credentials are excluded from the repository.

Edit the `settings.py` file of your project and add the following code to it:

```
from decouple import config
# Email server configuration

EMAIL_HOST = 'smtp.gmail.com'

EMAIL_HOST_USER = config('EMAIL_HOST_USER')

EMAIL_HOST_PASSWORD = config('EMAIL_HOST_PASSWORD')

EMAIL_PORT = 587

EMAIL_USE_TLS = True

DEFAULT_FROM_EMAIL = config('DEFAULT_FROM_EMAIL')
```

The `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD` and `DEFAULT_FROM_EMAIL` settings are now loaded from environment variables defined in the `.env` file.

Instead of Gmail, you can also use a professional, scalable email service that allows you to send emails via SMTP using your own domain, such as SendGrid (<https://sendgrid.com/>) or Amazon Simple Email Service (SES) (<https://aws.amazon.com/ses>).

Both services will require you to verify your domain and sender email accounts and will provide you with SMTP credentials to send emails. The `django-anymail` application simplifies the task of adding email service providers to your project like SendGrid or Amazon SES.

If you **can't use an SMTP server**, you can tell Django to write emails to the console by adding the following setting to the `settings.py` file:

```
'EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

By using this setting, Django will output all emails to the shell instead of sending them. This is very useful for testing your application without an SMTP server.

In order to send emails with Gmail's SMTP server, make sure that two-step verification is active in your Gmail account.

الفرق الرئيسي بينهما		الجانب
Console Backend	SMTP Backend	
لا، يطبع الرسالة في الـ Console فقط	نعم، يرسل البريد إلى المستلمين	الإرسال الفعلي
التطوير والاختبار المحلي	الإنتاج أو الاختبار مع خادم حقيقي	الاستخدام
لا يتطلب أي إعداد إضافي	يتطلب إعداد خادم SMTP وبيانات اعتماد	الإعداد
نص الرسالة يظهر في الطرفية	بريد إلكتروني يصل إلى صندوق الوارد	النتج

Open <https://myaccount.google.com/security> in your browser and enable 2-Step Verification for your account.

How you sign in to Google

Make sure that you can always access your Google Account by keeping this information up to date

🛡️ 2-Step Verification

✅ On since 1 Jan 2024

Then, you need to create an app password and use it for your SMTP credentials. An app password is a 16-digit passcode that gives a less secure app or device permission to access your Google account.

To create an app password, open <https://myaccount.google.com/apppasswords> in your browser.

You don't have any app passwords.

To create a new app specific password, type a name for it below...

App name

If you cannot access App passwords, it **might be that 2-Step Verification is not set for your account, your account is an organization account instead of a standard Gmail account, or you turned on Google's advanced protection**. Make sure to use a standard Gmail account and activate 2-Step Verification for your Google account.

Enter the name Blog and click the Create button, as follows:

You don't have any app passwords.

To create a new app specific password, type a name for it below...



A new password will be generated and displayed like this:



Copy the generated app password, **edit the .env file** of your project and add the app password to the EMAIL_HOST_PASSWORD variable, as follows:

EMAIL_HOST_USER=your_account@gmail.com

EMAIL_HOST_PASSWORD=xxxxxxxxxxxxxxxxxx

DEFAULT_FROM_EMAIL=My Blog <your_account@gmail.com>

Open the Python shell by running the following command in the system shell prompt: **python manage.py shell**

Execute the following code in the Python shell:

from django.core.mail import send_mail <<<

,'.send_mail('Django mail', 'This e-mail was sent with Django <<<

,your_account@gmail.com', ['your_account@gmail.com'])

(fail_silently=False

The send_mail() function takes the subject, message, sender, and list of recipients as required arguments. By setting the optional argument fail_silently=False, we are telling it to raise an exception if the email cannot be sent.

If the output you see is 1, then your email was successfully sent.

1. `fail_silently=False` (الافتراضي إذا لم تُحدد القيمة في بعض الحالات):

- إذا فشل إرسال البريد الإلكتروني (مثل مشكلة في الاتصال بخادم SMTP، بيانات اعتماد غير صحيحة، أو مشكلة في العنوان)، فإن Django سيرفع استثناء (مثل `smtpplib.SMTPException`).
- هذا مفيد عندما تريد أن تعرف فوراً إذا كان هناك خطأ ما في عملية الإرسال، خاصة أثناء الاختبار أو تصحيح الأخطاء (debugging).
- مثال على الاستثناء:

Copy Wrap Expand

hidden lines 2

2. `fail_silently=True`:

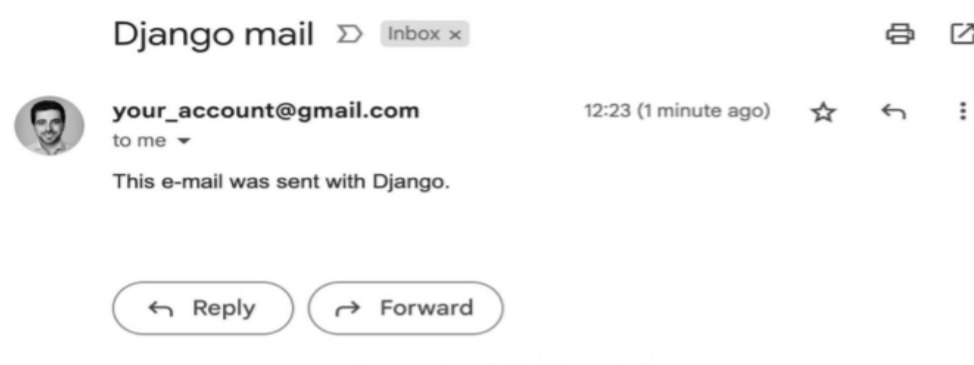
- إذا فشل إرسال البريد الإلكتروني، فإن Django لن يرفع استثناء وسيجاهل الخطأ بهدوء.
- الدالة ستُرجع 0 (للإشارة إلى الفشل) بدلاً من 1، لكن التطبيق سيستمر في العمل دون توقف.
- هذا الخيار مفيد في بيئة الإنتاج إذا كنت لا تريد أن يتوقف التطبيق بسبب مشكلة في إرسال البريد الإلكتروني (مثل إشعار غير حاسم)

If you get a `CERTIFICATE_VERIFY_FAILED` error, install the `certify` module with the command `pip install --upgrade certify`

If you are using macOS, run the following command on the shell to install `certify` and let Python access macOS root certificates:

`Applications/Python\ 3.12/Install\ Certificates.command/`

Check your inbox. You should have received the email



Sending emails in views

Edit the `post_share view` in the `views.py` file of the blog application, as follows:

`from django.core.mail import send_mail`

`... #`

```

def post_share(request, post_id):
    Retrieve post by id #
    post = get_object_or_404( Post, id=post_id, status=Post.Status.PUBLISHED )
    sent = False
    if request.method == 'POST':
        Form was submitted #
        form = EmailPostForm(request.POST)
        if form.is_valid():
            Form fields passed validation #

            cd = form.cleaned_data

            post_url = request.build_absolute_uri( post.get_absolute_url() )
            subject = ( f'{cd["name"]} ({cd["email"]})
            ( "f"recommends you read {post.title}

            "message = ( f"Read {post.title} at {post_url}\n\n
            ( "f"{cd["name"]}'s comments: {cd["comments"]}'
            ,send_mail( subject=subject, message=message
            ( from_email=None, recipient_list=[cd["to"]]

            sent = True

        else:

            form = EmailPostForm

    return render( request, 'blog/post/share.html', { 'post': post, 'form': form, 'sent
    ( { sent

```

we have declared a sent variable with the initial False value. We set this variable to True after the email is sent. We will use the sent variable in the template to display a success message when the form is successfully submitted.

Since we have to include a link to the post in the email, we retrieve the absolute path of the post using its `get_absolute_url()` method. We use this path as an input for `request.build_absolute_uri()` to build a complete URL, including the HTTP schema and hostname.

- `request.build_absolute_uri()` تحول مسارًا نسبيًا إلى URL مطلق كامل بناءً على سياق الطلب.
- فائدتها: تُنتج روابط قابلة للمشاركة تعمل في أي بيئة، مثل البريد الإلكتروني في مثالك.
- استخدامها في الكود: تضمن أن المستلم يتلقى رابطًا عمليًا للمنشور بدلاً من مسار نسبي غير مفيد.

We create the subject and the message body of the email using the cleaned data of the validated form.

Finally, we send the email to the email address contained in the to field of the form. In the from_email parameter, we pass the None value, so the value of the DEFAULT_FROM_EMAIL setting will be used for the sender.

Now that the view is complete, we have to add a new URL pattern for it.

Open the urls.py file of your blog application and add the post_share URL pattern, as follows:

```
path('<int:post_id>/share/', views.post_share, name='post_share'),
```

Rendering forms in templates

Create a new file in the **blog/templates/blog/post/ directory** and name it share.html.

Add the following code to the new **share.html** template:

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

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

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

```
{% if sent %}
```

```
<h1>E-mail successfully sent</h1>
```

```
<p> "{{ post.title }}" was successfully sent to {{ form.cleaned_data.to }}.</p>
```

```
{% else %}
```

```
<h1>Share "{{ post.title }}" by e-mail</h1>
```

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

```
{{ form.as_p }}
```

```
{% csrf_token %}
```

```
<"input type="submit" value="Send e-mail>
```

```
<form/>
```

```
{% endif %}
```

```
{% endblock %}
```


This is the template that is used to both display the form to share a post via email and to display a success message when the email has been sent. We differentiate between both cases with {% if sent %}.

To display the form, we have defined an HTML form element, indicating that it has to be submitted by the POST method: **<form method="post">**

We have included the form instance with {{ form.as_p }}. We tell Django to render the form fields using HTML paragraph <p> elements by using the as_p method. We could also render the form as an unordered list with as_ul or as an HTML table with as_table.

We have added a {% csrf_token %} template tag. This tag introduces a hidden field with an autogenerated token to avoid cross-site request forgery (CSRF) attacks. These attacks consist of a malicious website or program performing an unwanted action for a user on the site.

The {% csrf_token %} template tag generates a hidden field that is rendered like this: **'input type="hidden" name="csrfmiddlewaretoken" value="26JjKo2lcEtYkGoV9z4XmJIEHLXN5LDR" />**

Notice:

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

Edit the blog/post/detail.html template and make it look like this:

```
#####

<p class="date">Published {{ post.publish }} by {{ post.author }} </p>

{{ post.body|linebreaks }}

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

{% endblock %}
```

We have added a link to the post_share URL. The URL is built dynamically with the {% url %} template tag provided by Django. We use the namespace called blog and the URL named post_share. We pass the id post as a parameter to build the URL.

Open the shell prompt and execute the following command to start the development server then Open **http://127.0.0.1:8000/blog/** in your browser and click on any post title to view the post detail page. **python manage.py runserver**

Notes on Duke Ellington

Published Jan. 3, 2024, 1:19 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](#)

Figure 2.16: The post detail page, including a link to share the post

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

Share "Notes on Duke Ellington" by e-mail

Name:

Email:

To:

Comments:

[SEND E-MAIL](#)

My blog

This is my blog.

When you click on the SEND E-MAIL button, the form is submitted and validated. If all fields contain valid data, you get a success message.

E-mail successfully sent

"Notes on Duke Ellington" was successfully sent to your_account@gmail.com.

My blog
This is my blog.

Figure 2.18: A success message for a post shared via email

Send a post to your own email address and check your inbox. The email you receive should look like this:



Figure 2.19: Test email sent displayed in Gmail

If you submit the form with invalid data, the form will be rendered again, including all validation errors:

Share "Notes on Duke Ellington" by e-mail

Name:

Email:

To:

Comments:

[SEND E-MAIL](#)

My blog

This is my blog.

• Enter a valid email address.

• This field is required.

Most modern browsers will prevent you from submitting a form with empty or erroneous fields. This is because the browser validates the fields based on their attributes before submitting the form. In this case, the form won't be submitted, and the browser will display an error message for the fields that are wrong.

To test the Django form validation using a modern browser, you can skip the browser form validation by adding the `novalidate` attribute to the HTML `form` element, like `<form method="post" novalidate>`.

You can add this attribute to prevent the browser from validating fields and test your own form validation. After you are done testing, remove the `novalidate` attribute to keep the browser form validation.

Creating a comment system

comment system will allow users to comment on posts. To build the comment system, we will need the following:

- A comment model to store user comments on posts
- A Django form that allows users to submit comments and manages the data validation
- A view that processes the form and saves a new comment to the database
- A list of comments and the HTML form to add a new comment that can be included in the post detail template

Open the `models.py` file of your blog application and add the following code:

```
class Comment(models.Model):
    ,post = models.ForeignKey( Post, on_delete=models.CASCADE
    ( 'related_name='comments

    name = models.CharField(max_length=80)

    )email = models.EmailField

    )body = models.TextField

    created = models.DateTimeField(auto_now_add=True)

    updated = models.DateTimeField(auto_now=True)

    active = models.BooleanField(default=True)
```

```

class Meta:

    ordering = ['created']

    indexes = [models.Index(fields=['created']), ]

    def __str__(self):

        'return f'Comment by {self.name} on {self.post}'

```

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

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

We have defined the active Boolean field to control the status of the comments. This field will allow us to manually deactivate inappropriate comments using the administration site. We use **default=True** to indicate that all comments are active by default.

We have defined the created field to store the date and time when the comment was created. By using **auto_now_add**, the date will be saved automatically when creating an object. In the Meta class of the model, we have added **ordering = ['created']** to sort comments in chronological order by default, and we have added an index for the created field in ascending order. This will improve the performance of database lookups or ordering results using the created field.

The Comment model that we have built is not synchronized with the database. We need to generate a new database migration to create the corresponding database table.

Run the following command from the shell prompt:

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

We need to create the related database schema and apply the changes to the database.

Run the following command to apply existing migrations:

```
python manage.py migrate
```

Adding comments to the administration site

we will add the new model to the administration site to manage comments through a simple interface.

Open the admin.py file of the blog application, import the Comment model, and add the following ModelAdmin class:

```
from .models import Comment, Post
```

```
admin.register(Comment)@
```

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

```
list_display = ['name', 'email', 'post', 'created', 'active']
```

```
list_filter = ['active', 'created', 'updated']
```

```
search_fields = ['name', 'email', 'body']
```

Open the shell prompt and execute the following command to start the development server, Open <http://127.0.0.1:8000/admin/> in your browser.

```
python manage.py runserver
```

Creating forms from models

We need to build a form to let users comment on blog posts. Remember that Django has two base classes that can be used to create forms: Form and ModelForm. We used the Form class to allow users to share posts by email. Now, we will use ModelForm to take advantage of the existing Comment model and build a form dynamically for it.

Edit the forms.py file of your blog application and add the following lines:

```
from .models import Comment
```

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

```
:class Meta
```

```
model = Comment
```

```
fields = ['name', 'email', 'body']
```

To create a form from a model, we just indicate which model to build the form for in the Meta class of the form. Django will introspect the model and build the corresponding form dynamically.

Each model field type has a corresponding default form field type. The attributes of model fields are taken into account for form validation. By default, Django creates a form field for each field contained in the model. However, we can explicitly tell Django which fields to include in the form using the **fields attribute** or define which fields to exclude using the **exclude attribute**. In the **CommentForm** form, we have explicitly included the name, email, and body fields. These are the only fields that will be included in the form.

Handling ModelForms in views

For sharing posts by email, we used the same view to display the form and manage its submission. We used the HTTP method to differentiate between both cases: GET to display the form and POST to submit it. In this case, we will add the comment form to the post detail page, and we will build a separate view to handle the form submission. The new view that processes the form will allow the user to return to the post detail view once the comment has been stored in the database.

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

```
from django.views.decorators.http import require_POST

from .forms import CommentForm, EmailPostForm

#####

from .models import Post

#####

require_POST@
def post_comment(request, post_id):

    post = get_object_or_404( Post, id=post_id,status=Post.Status.PUBLISHED)

    comment = None

    A comment was posted #

    form = CommentForm(data=request.POST)

    :()if form.is_valid

    Create a Comment object without saving it to the database #

    comment = form.save(commit=False)
```

Assign the post to the comment #

```
comment.post = post
```

Save the comment to the database #

```
()comment.save
```

```
:return render( request, 'blog/post/comment.html', { 'post': post, 'form  
({ form,'comment': comment
```

We have defined the `post_comment` view that takes the request object and the `post_id` variable as parameters. We will be using this view to manage the post submission. We expect the form to be submitted using the HTTP POST method. **We use the `require_POST` decorator provided by Django to only allow POST requests for this view.**

Django allows you to restrict the HTTP methods allowed for views. Django will throw an HTTP 405 (method not allowed) error if you try to access the view with any other HTTP method.

we have implemented the following actions:

- ()We retrieve a published post by its id using the `get_object_or_404` shortcut
- We define a comment variable with the initial value `None`. This variable will be used to store the comment object when it is created.
- We instantiate the form using the submitted POST data and validate it using the `is_valid()` method. If the form is invalid, the template is rendered with the validation errors.
- If the form is valid, we create a new `Comment` object by calling the form's `save()` method and assign it to the comment variable, as follows:

```
comment = form.save(commit=False)
```

- The `save()` method creates an instance of the model that the form is linked to and saves it to the database. If you call it using `commit=False`, the model instance is created but not saved to the database. This allows us to modify the object before finally saving it.
- We assign the post to the comment we created:

```
comment.post = post
```

- We save the new comment to the database by calling its `save()` method:
`comment.save()`
- ,We render the `blog/post/comment.html` template, passing the `post`, `form` and `comment` objects in the template context. This template doesn't exist yet; we will create it.

Notice:

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

Edit the `urls.py` file of the blog application and add the following URL pattern to it:

```
path( '<int:post_id>/comment/', views.post_comment, name='post_comment' ),
```

We have implemented the view to manage the submission of comments and their corresponding URL.

Creating templates for the comment form

We will create a template for the comment form that we will use in two places:

- In the post detail template associated with the `post_detail` view to let users publish comments.
- In the post comment template associated with the `post_comment` view to display the form again if there are any form errors.

We will create the form template and use the `{% include %}` template tag to include it in the two other templates.

In the `templates/blog/post/` directory, create a new `includes/` directory. Add a new file inside this directory and name it `comment_form.html`.

Edit the new `blog/post/includes/comment_form.html` template and add the following code:

```
<h2>Add a new comment</h2>

<"form action="{% url "blog:post_comment" post.id %}" method="post>

{{ form.as_p }}

{% csrf_token %}

<p><input type="submit" value="Add comment"></p>

</form/>
```


In this template, we build the action URL of the HTML `<form>` element dynamically using the `{% url %}` template tag. We build the URL of the `post_comment` view that will process the form. We display the form rendered in paragraphs and we include `{% csrf_token %}` for CSRF protection because this form will be submitted with the POST method

Create a new file in the `templates/blog/post/` directory of the blog application and name it `comment.html`

Edit the new `blog/post/comment.html` template and add the following code:

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

{% block title %}Add a comment{% endblock %}

{% block content %}

{% if comment %}

<h2>Your comment has been added.</h2>

<p><a href="{{ post.get_absolute_url }}">Back to the post</a></p>

{% else %}

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

{% endif %}

{% endblock %}
```

The template covers two different scenarios:

- If the form data submitted is valid, the `comment` variable will contain the comment object that was created and a success message will be displayed.
- If the form data submitted is not valid, the `comment` variable will be `None`. In this case, we will display the comment form. We use the `{% include %}` template tag to include the `comment_form.html` template that we have previously created.

Adding comments to the post detail view

To complete the comment functionality, we will add the list of comments and the comment form to the `post_detail` view.

Edit the views.py file of the blog application and edit the post_detail view as follows:

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

Let's review the code we have added to the post_detail view:

- ,We have added a QuerySet to retrieve all active comments for the post as follows:

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

- This QuerySet is built using the post object. Instead of building a QuerySet for the Comment model directly, we leverage the post object to retrieve the related Comment objects. We use the comments manager for the related Comment objects that we previously defined in the Comment model, using the related_name attribute of the ForeignKey field to the Post model.
- We have also created an instance of the comment form with

form = CommentForm().

Adding comments to the post detail template

We need to edit the blog/post/detail.html template to implement the following:

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

We will start by adding the total number of comments for a post.

Edit the blog/post/detail.html template and change it as follows:

```
#####
```

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

```
{% with comments.count as total_comments %}
```

```
<h2>
```

```
{{ total_comments }} comment{{ total_comments|pluralize }}
```

```
<h2/>
```

```
{% endwith %}
```

```
{% endblock %}
```

,We use the Django object relational mapper (ORM) in the template executing the comments.count() QuerySet. Note that the Django template language doesn't use parentheses for calling methods.

The {% with %} tag allows you to assign a value to a new variable that will be available in the template until the {% endwith %} tag.

Note:

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

We use the pluralize template filter to display a plural suffix for the word "comment," depending on the total_comments value. Template filters take the value of the variable they are applied to as their input and return a computed value.

The pluralize template filter returns a string with the letter "s" if the value is different from 1. The preceding text will be rendered as 0 comments, 1 comment, or N comments, depending on the number of active comments for the post.

add the list of active comments to the post detail template.

Edit the blog/post/detail.html template and implement the following changes:

```
#####
```

```

{% for comment in comments %}

<div class="comment">

<p class="info">

Comment {{ forloop.counter }} by {{ comment.name }}

{{ comment.created }}
<p/>

{{ comment.body|linebreaks }}

</div>

{% empty %}

<p>There are no comments.</p>

{% endfor %}

{% endblock %}

```

We have added a `{% for %}` template tag to loop through the post comments. If the comments list is empty, we display a message that informs users that there are no comments for this post. We enumerate comments with the `{{ forloop.counter }}` variable, which contains the loop counter in each iteration. For each post, we display the name of the user who posted it, the date, and the body of the comment.

let's add the comment form to the template.

Edit the `blog/post/detail.html` template and include the comment form template as follows:

```

{% empty %}

<p>There are no comments.</p>

{% endfor %}

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

{% endblock %}

```

Open `http://127.0.0.1:8000/blog/` in your browser and click on a post title to take a look at the post detail page.

Notes on Duke Ellington

Published Jan. 3, 2024, 1:19 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](#)

0 comments

There are no comments yet.

Add a new comment

Name:

Email:

Body:

ADD COMMENT

My blog

This is my blog.

Fill in the comment form with valid data and click on Add comment. You should see the following page:

Fill in the comment form with valid data and click on **Add comment**. You should see the following page:

Your comment has been added.

[Back to the post](#)

My blog

This is my blog.

Click on the Back to the post link. You should be redirected back to the post detail page, and you should be able to see the comment that you just added, as follows:

Notes on Duke Ellington

Published Jan. 3, 2024, 1:19 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](#)

1 comment

Comment 1 by Antonio Jan. 4, 2024, 4:07 p.m.

I didn't know that!

Add a new comment

Name:

Email:

Body:

ADD COMMENT

My blog

This is my blog.

Add one more comment to the post. The comments should appear below the post contents in chrono-logical order, as follows:

2 comments

Comment 1 by Antonio Jan. 4, 2024, 4:07 p.m.

I didn't know that!

Comment 2 by Bienvenida Jan. 4, 2024, 4:11 p.m.

I really like this article.

Figure 2.26: The comments list on the post detail page.

Open <http://127.0.0.1:8000/admin/blog/comment/> in your browser. You will see the administration page with the list of comments you created, like this:

Select comment to change

Q Search

Action: Go 0 of 2 selected

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	ACTIVE
<input type="checkbox"/>	Antonio	your_account@gmail.com	Notes on Duke Ellington	Jan. 4, 2024, 4:07 p.m.	<input checked="" type="checkbox"/>
<input type="checkbox"/>	Bienvenida	bienvenida@example.com	Notes on Duke Ellington	Jan. 4, 2024, 4:11 p.m.	<input checked="" type="checkbox"/>

2 comments




Figure 2.27: List of comments on the administration site

Click on the name of one of the posts to edit it. Uncheck the **Active** checkbox as follows and click on the **Save** button:

Chapter 2 101

Change comment

Comment by Antonio on Notes on Duke Ellington HISTORY

Post:   

Name:


Email:

Body:

☐ Active

Figure 2.28: Editing a comment on the administration site

You will be redirected to the list of comments. The **Active** column will display an inactive icon for the comment, as shown in Figure 2.29:

 The comment "Comment by Antonio on Notes on Duke Ellington" was changed successfully.

Select comment to change

Q Search

Action: Go 0 of 2 selected

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	ACTIVE
<input type="checkbox"/>	Antonio	your_account@gmail.com	Notes on Duke Ellington	Jan. 4, 2024, 4:07 p.m.	<input type="checkbox"/>
<input type="checkbox"/>	Bienvenida	bienvenida@example.com	Notes on Duke Ellington	Jan. 4, 2024, 4:11 p.m.	<input checked="" type="checkbox"/>

If you return to the post detail view, you will note that the inactive comment is no longer displayed, neither is it counted for the total number of active comments for the post

1 comment

Comment 1 by Bienvenida Jan. 4, 2024, 4:11 p.m.

I really like this article.

Using simplified templates for form rendering

You have used `{{ form.as_p }}` to render the forms using HTML paragraphs. This is a very straight-forward method for rendering forms, but there may be occasions when you need to employ custom HTML markup for rendering forms.

To use custom HTML for rendering form fields, you can access each form field directly, or iterate through the form fields, as in the following example:

```
{% for field in form %}
<div class="my-div">
  {{ field.errors }}
  {{ field.label_tag }} {{ field }}
  <div class="help-text">{{ field.help_text|safe }}</div>
</div>
{% endfor %}
```

In this code, we use `{{ field.errors }}` to render any field errors of the form, `{{ field.label_tag }}` to render the form HTML label, `{{ field }}` to render the actual field, and `{{ field.help_text|safe }}` to render the field's help text HTML.

This method is helpful to customize how forms are rendered, but you might need to add certain HTML elements for specific fields or include some fields in containers. Django 5.0 introduces field groups and field group templates. Field groups simplify the rendering of labels, widgets, help texts, and field errors. Let's use this new feature to customize the comment form.

We are going to use custom HTML markup to reposition the name and email form fields using additional HTML elements.

Edit the `blog/post/includes/comment_form.html` template and modify it as follows. The new code is highlighted in bold:

```
<h2>Add a new comment</h2>  
<form action="{% url 'blog:post_comment' post.id %}" method='post>  
<div class='left>  
{{ form.name.as_field_group }}  
</div>  
<div class='left>  
{{ form.email.as_field_group }}  
</div>  
{{ form.body.as_field_group }}  
{% csrf_token %}  
<p><input type='submit' value='Add comment'></p>  
</form>
```

We have added `<div>` containers for the name and email fields with a custom CSS class to float both fields to the left. The `as_field_group` method renders each field including help text and errors. This method uses the `django/forms/field.html` template by default.

You can also create custom field templates and reuse them by adding the `template_name` attribute to any form field.



Add a new comment

Name: Email:

Body:

ADD COMMENT