2

Усовершенствование блога за счет продвинутых функциональностей

В предыдущей главе мы ознакомились с главными компонентами Django, разработав простое приложение для ведения блога. Мы создали простое приложение `blog`, используя представления, шаблоны и URL-адреса. В этой главе мы расширим функциональности приложения `blog` за счет функций, которые в настоящее время можно найти на многих блоговых платформах.

В данной главе будут рассмотрены следующие темы:

- использование канонических URL-адресов для моделей;
- создание дружественных для поисковой оптимизации URL-адресов постов;
- добавление постраничной разбивки в представление списка постов;
- разработка представлений на основе классов;
- отправка электронных писем с помощью Django;
- использование форм Django, позволяющих делиться постами по электронной почте;
- добавление комментариев к постам с использованием форм из моделей.

Исходный код к этой главе находится по адресу https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Использование канонических URL-адресов для моделей

На веб-сайте могут быть разные страницы, отображающие один и тот же контент. В нашем приложении изначальная часть контента по каждому посту отображается как на странице списка постов, так и на странице детальной информации о посте. Канонический URL-адрес – это предпочтительный URL-адрес ресурса. Его можно представить как URL-адрес наиболее репрезентативной страницы с конкретным контентом. На сайте могут быть разные страницы, которые показывают посты, но есть один URL-адрес, который используется в качестве главного URL-адреса поста. Канонические URL-адреса позволяют указывать URL-адрес мастер-копии страницы. Django дает возможность в своих собственных моделях реализовывать метод `get_absolute_url()`, который возвращает канонический URL-адрес объекта.

Мы будем использовать URL-адрес `post_detail`, определенный в шаблонах URL-адресов приложения, чтобы формировать канонический URL-адрес для объектов `Post`. Django предоставляет различные функции-резольверы URL-flhtcjd[1], которые позволяют формировать URL-адреса динамически, используя их имя и любые требуемые параметры. Мы будем использовать функцию-утилиту `reverse()`[2] модуля `django.urls`.

Отредактируйте файл `models.py` приложения `blog`, импортировав функцию `reverse()` и добавив метод `get_absolute_url()` в модель `Post`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
from django.urls import reverse

class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
                      .filter(status=Post.Status.PUBLISHED)

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
```

[1] Резольвер URL-адресов – это программная утилита или функция, которая конвертирует логический адрес или метаданные в физический URL-адрес целевых данных. – *Прим. перев.*

[2] URL-адрес, или URL-указатель (от англ. Uniform Resource Locator, аббр. URL, т. е. Унифицированный указатель ресурса), указывает на ресурс в сети. Функция `reverse` в Django выполняет обратное действие и используется для отыскания URL-адреса / URL-указателя заданного ресурса. – *Прим. перев.*

```
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

    def __str__(self):
        return self.title

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

Функция `reverse()` будет формировать URL-адрес динамически, применяя имя URL-адреса, определенное в шаблонах URL-адресов. Мы использовали именное пространство `blog`, за которым следуют двоеточие и URL-адрес `post_detail`. Напомним, что именное пространство `blog` определяется в главном файле `urls.py` проекта при вставке шаблонов URL-адресов из `blog.urls`. URL-адрес `post_detail` определен в файле `urls.py` приложения `blog`. Результирующий строковый литерал, `blog:post_detail`, можно использовать глобально в проекте, чтобы ссылаться на URL-адрес детальной информации о посте. Этот URL-адрес имеет обязательный параметр – `id` извлекаемого поста блога. Идентификатор `id` объекта `Post` был включен в качестве позиционного аргумента, используя параметр `args=[self.id]`.

Подробнее о функциях-резольверах URL-адресов можно узнать на странице https://docs.djangoproject.com/en/4.1/ref/urlresolvers/.

Давайте заменим URL-адреса детальной информации о посте в шаблонах новым методом `get_absolute_url()`.

Отредактируйте файл `blog/post/list.html`, заменив строку

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

строкой

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

Теперь файл blog/post/list.html должен выглядеть следующим образом:

```
{% 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 %}
{% endblock %}
```

Откройте оболочку и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу http://127.0.0.1:8000/blog/ в своем браузере. Ссылки на одиночные посты блога по-прежнему должны работать. Теперь Django формирует их, используя метод get_absolute_url() модели Post.

Создание дружественных для поисковой оптимизации URL-адресов постов

Канонический URL-адрес представления детальной информации о посте блога в настоящее время выглядит как /blog/1/. Мы изменим шаблон URL-адреса, чтобы формировать дружественные для поисковой оптимизации URL-адреса постов. В целях формирования URL-адресов одиночных постов

мы будем использовать дату публикации `publish` и значения `slug`. Присоединив даты, мы приведем URL-адрес детальной информации о посте к следующему виду: `/blog/2022/1/1/who-was-django-reinhardt/`. Мы предоставим поисковым механизмам дружественные для индексации URL-адреса, содержащие как заголовок, так и дату поста.

Для того чтобы получить одиночные посты с комбинацией даты публикации и слага, необходимо обеспечить, чтобы ни одну запись невозможно было сохранить в базе данных с тем же значением поля `slug` и поля `publish`, что и у существующего поста. Мы предотвратим хранение в модели `Post` дублирующихся записей, определив, что слаги являются уникальными для даты публикации поста.

Отредактируйте файл `models.py`, добавив следующий ниже параметр `unique_for_date` в поле `slug` модели `Post`:

```
class Post(models.Model):
    # ...
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    # ...
```

Теперь при использовании параметра `unique_for_date` поле `slug` должно быть уникальным для даты, сохраненной в поле `publish`. Обратите внимание, что поле `publish` является экземпляром класса `DateTimeField`, но проверка на уникальность значений будет выполняться только по дате (не по времени). Django будет предотвращать сохранение нового поста с тем же именем, что и у существующего поста на заданную дату публикации. В результате мы обеспечили уникальность слагов для даты публикации, поэтому теперь можно извлекать одиночные посты по полям `publish` и `slug`.

Мы изменили модели, поэтому давайте создадим миграции. Обратите внимание, что параметр `unique_for_date` не соблюдается на уровне базы данных, поэтому миграция базы данных не требуется. Между тем миграции в Django используются для отслеживания всех изменений модели. Мы создадим миграцию только для того, чтобы привести миграции в соответствие с текущим состоянием модели.

Выполните следующую ниже команду в командной оболочке:

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

Вы должны получить следующий ниже результат:

```
Migrations for 'blog':
    blog/migrations/0002_alter_post_slug.py
    - Alter field slug on post
```

Django только что создал файл `0002_alter_post_slug.py` внутри каталога `migrations` приложения `blog`.

Выполните следующую ниже команду в командной оболочке, чтобы применить существующие миграции:

```
python manage.py migrate
```

Вы получите результат, который заканчивается такой строкой:

```
Applying blog.0002_alter_post_slug... OK
```

Django будет считать, что все миграции были применены и модели синхронизированы. В базе данных не будет выполнено никаких действий, поскольку параметр `unique_for_date` не применяется на уровне базы данных.

Видоизменение шаблонов URL-адресов

Давайте видоизменим шаблоны URL-адресов, чтобы использовать дату публикации и слаг для URL-адреса детальной информации о посте.

Отредактируйте файл `urls.py` приложения `blog`, заменив строку

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

строками

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

Теперь файл `urls.py` должен выглядеть следующим образом:

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

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
```

```
         name='post_detail'),
]
```

Шаблон URL-адреса представления `post_detail` принимает следующие ниже аргументы:

- `year`: требуется целое число;
- `month`: требуется целое число;
- `day`: требуется целое число;
- `post`: требуется слаг (строка, содержащая только буквы, цифры, знаки подчеркивания или дефисы).

Конвертор пути `int` используется для параметров `year`, `month` и `day`, тогда как конвертор пути `slug` применяется для параметра `post`. В предыдущей главе вы узнали о конверторах путей. Все предоставляемые фреймворком Django конверторы путей можно посмотреть на странице https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters.

Видоизменение представлений

Теперь необходимо видоизменить параметры представления `post_detail`, чтобы они соответствовали новым параметрам URL-адреса, и использовать их для извлечения соответствующего объекта `Post`.

Откройте файл `views.py` и отредактируйте представление `post_detail`, как показано ниже:

```
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})
```

Мы видоизменили представление `post_detail`, чтобы использовать аргументы `year`, `month`, `day` и `post` и извлекать опубликованный пост с заданным слагом и датой публикации. Ранее, добавив в поле `slug` значение параметра `unique_for_date='publish'` модели `Post`, мы обеспечили, чтобы был только один пост со слагом на заданную дату. Таким образом, используя дату и слаг, можно извлекать одиночные посты.

Видоизменение канонического URL-адреса постов

Также необходимо видоизменить параметры канонического URL-адреса для постов блога, чтобы они сочетались с новыми параметрами URL-адреса.

Откройте файл `models.py` приложения `blog` и отредактируйте метод `get_absolute_url()`, как показано ниже:

```
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])
```

Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

Далее можно вернуться в свой браузер и кликнуть по одному из заголовков постов, чтобы посмотреть детальную информацию о посте. Вы должны увидеть что-то вроде этого:

Рис. 2.1. Страница представления детальной информации о посте

Взгляните на URL-адрес – он должен выглядеть как `/blog/2022/1/1/who-was-django-reinhardt/`. Вы разработали дружественные для поисковой оптимизации URL-адреса постов блога.

Добавление постраничной разбивки

Когда вы начнете добавлять контент в свой блог, вы сможете легко хранить десятки и даже сотни постов в своей базе данных. Возможно, вы захотите разделить список постов на несколько страниц, не отображая все записи на одной странице, и вставить навигационные ссылки на разные страницы. Эта функциональность называется постраничной разбивкой, и ее можно найти почти в каждом веб-приложении, которое показывает длинные списки элементов.

Например, Google использует постраничную разбивку с целью распределения результатов поиска по нескольким страницам. На рис. 2.2 показаны постранично разбитые ссылки Google на страницы результатов поиска:

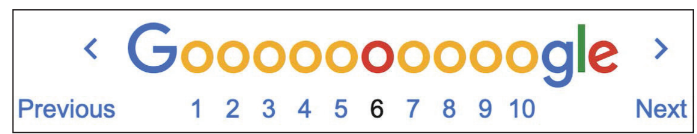

Рис. 2.2. Постранично разбитые ссылки Google на страницы результатов поиска

В Django есть встроенный класс постраничной разбивки, который позволяет легко управлять постранично разбитыми данными, при этом имеется возможность определять число объектов, которое необходимое возвращать в расчете на страницу, и извлекать записи, соответствующие запрошенной пользователем странице.

Добавление постраничной разбивки в представление списка постов

Отредактируйте файл `views.py` приложения `blog`, импортировав класс Django `Paginator` и видоизменив представление `post_list`, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator

def post_list(request):
    post_list = Post.published.all()
    # Постраничная разбивка с 3 постами на страницу
    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})
```

Давайте рассмотрим новый исходный код, который был добавлен в представление.

1. Мы создаем экземпляр класса `Paginator` с числом объектов, возвращаемых в расчете на страницу. Мы будем отображать по три поста на страницу.
2. Мы извлекаем HTTP `GET`-параметр `page` и сохраняем его в переменной `page_number`. Этот параметр содержит запрошенный номер страницы. Если параметра `page` нет в `GET`-параметрах запроса, то мы используем стандартное значение 1, чтобы загрузить первую страницу результатов.
3. Мы получаем объекты для желаемой страницы, вызывая метод `page()` класса `Paginator`. Этот метод возвращает объект `Page`, который хранится в переменной `posts`.
4. Мы передаем номер страницы и объект `posts` в шаблон.

Создание шаблона постраничной разбивки

Далее необходимо создать навигацию по страницам, чтобы пользователи имели возможность просматривать разные страницы. Мы создадим шаблон отображения постранично разбитых ссылок и сделаем его типовым, чтобы иметь возможность реиспользовать шаблон для постраничной разбивки любого объекта на веб-сайте.

Внутри каталога `templates/` создайте новый файл и назовите его `pagination.html`. Добавьте в файл следующий ниже исходный код HTML:

```
<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>
```

Это типовой шаблон постраничной разбивки. Предполагается, что данный шаблон будет иметь в контексте объект `Page`, чтобы прорисовывать преды-

дущую и следующую ссылки, а также отображать текущую страницу и общее число страниц результатов.

Давайте вернемся к шаблону `blog/post/list.html` и разместим шаблон `pagination.html` в нижней части блока `{% content %}`, как показано ниже:

```
{% 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 %}
```

Шаблонный тег `{% include %}` загружает данный шаблон и прорисовывает его с использованием текущего контекста шаблона. Ключевое слово `with` используется для того, чтобы передавать дополнительные контекстные переменные в шаблон. Для прорисовки в шаблоне постраничной разбивки используется переменная `page`, при этом объект `Page`, который мы передаем из представления в шаблон, называется `posts`. Мы используем выражение `with page=posts`, чтобы передавать переменную, ожидаемую шаблоном постраничной разбивки. Описанному методу можно следовать для применения шаблона постраничной разбивки для любого типа объекта.

Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/blog/post/` в своем браузере и используйте сайт администрирования, чтобы создать в общей сложности четыре разных поста. Проверьте, чтобы у всех этих постов был установлен статус **Published**.

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере. Вы должны увидеть первые три поста в обратном хронологическом порядке, а затем навигационные ссылки в нижней части списка постов, как показано ниже:

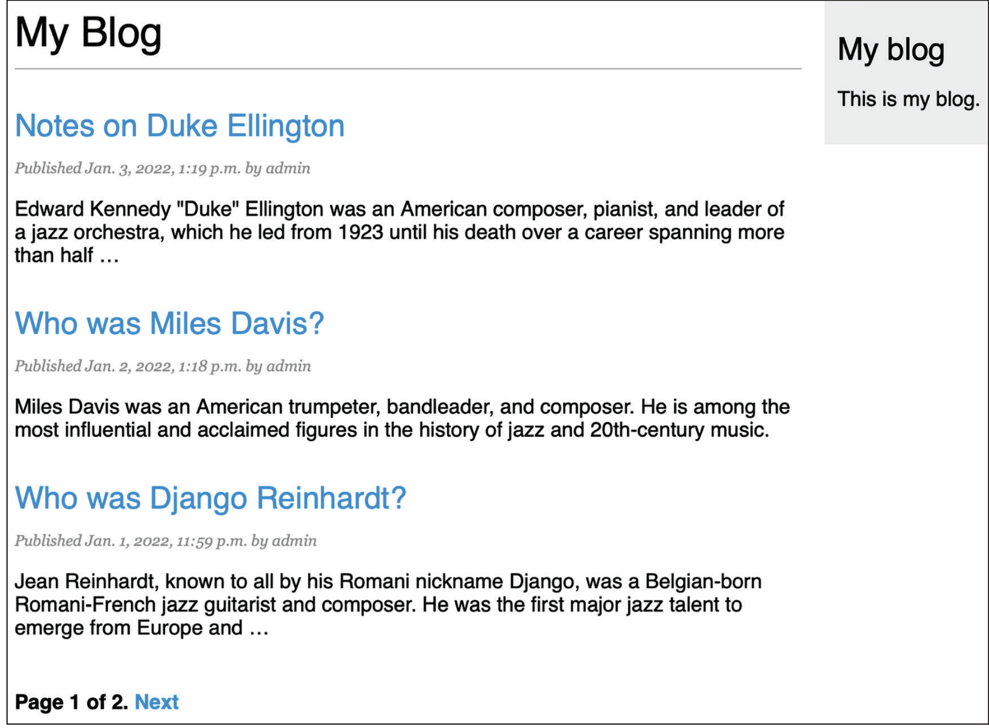

Рис. 2.3. Страница списка постов с постраничной разбивкой ссылок внизу

Если кликнуть по **Next** (Далее), то можно увидеть последний пост. URL-адрес второй страницы содержит `GET`-параметр `?page=2`. Указанный параметр используется представлением для загрузки запрошенной страницы результатов с использованием постраничного разбивщика.

Рис. 2.4. Вторая страница результатов

Отлично! Ссылки на постраничную разбивку работают так, как и ожидалось.

Обработка ошибок постраничной разбивки

Теперь, когда постраничная разбивка работает, в представление можно добавить обработку исключений, вызванных ошибками постраничной разбивки. Параметр `page`, используемый представлением для извлечения заданной страницы, потенциально может применяться с неправильными значениями, такими как несуществующие номера страниц или строковое значение, которое нельзя использовать в качестве номера страницы. Мы выполним соответствующую обработку ошибок для таких случаев.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=3` в своем браузере. Вы должны увидеть следующую ниже страницу с ошибкой:

EmptyPage at /blog/

That page contains no results

Request Method:	GET
Request URL:	http://127.0.0.1:8000/blog/?page=3
Django Version:	4.1
Exception Type:	EmptyPage
Exception Value:	That page contains no results
Exception Location:	/Users/amele/Documents/env/dbe4/lib/python3.10/site-packages/django/core/paginator.py, line 57, in validate_number
Raised during:	blog.views.post_list
Python Executable:	/Users/amele/Documents/env/dbe4/bin/python
Python Version:	3.10.6

Рис. 2.5. Страница с ошибкой EmptyPage

При извлечении страницы 3 объект `Paginator` выдает исключение `EmptyPage`, поскольку она находится вне диапазона.

Подлежащих отображению результатов нет. Давайте обработаем эту ошибку в представлении.

Отредактируйте файл `views.py` приложения `blog`, добавив необходимую инструкцию импорта и видоизменив представление `post_list`, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator, EmptyPage

def post_list(request):
    post_list = Post.published.all()
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except EmptyPage:
        # Если page_number находится вне диапазона, то
        # выдать последнюю страницу
```

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

Мы добавили блок `try` и `except`, чтобы при извлечении страницы управлять исключением `EmptyPage`. Если запрошенная страница находится вне диапазона, то мы возвращаем последнюю страницу результатов. Мы получаем общее число страниц посредством `paginator.num_pages`. Общее число страниц совпадает с номером последней страницы.

Снова пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=3` в своем браузере. Теперь исключение управляется представлением, и последняя страница результатов возвращается, как показано ниже.

Рис. 2.6. Последняя страница результатов

Данное представление также должно обрабатывать случай, когда в параметре `page` передается нечто отличное от целого числа.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=asdf` в своем браузере. Вы должны увидеть следующую ниже страницу ошибки:

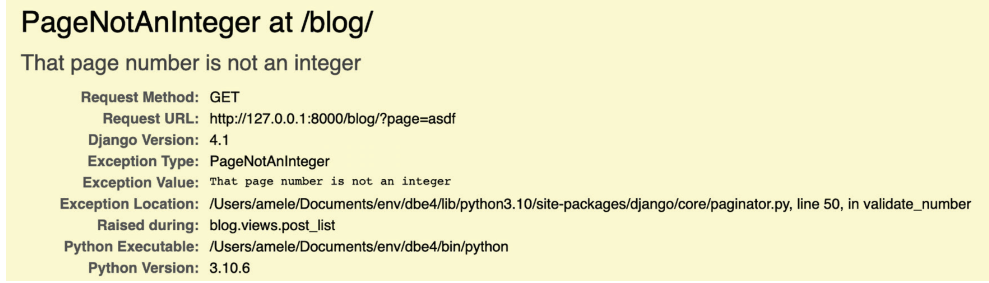

Рис. 2.7. Страница ошибки PageNotAnInteger

В этом случае при извлечении страницы `asdf` объект `Paginator` выдает исключение `PageNotAnInteger`, поскольку номера страниц могут быть только целыми числами. Давайте обработаем эту ошибку в представлении.

Отредактируйте файл `views.py` приложения `blog`, добавив необходимую инструкцию импорта и видоизменив представление `post_list`, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator, EmptyPage,\
                                  PageNotAnInteger
def post_list(request):
    post_list = Post.published.all()
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page')
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # Если page_number не целое число, то
        # выдать первую страницу
        posts = paginator.page(1)
    except EmptyPage:
        # Если page_number находится вне диапазона, то
        # выдать последнюю страницу результатов
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts})
```

Мы добавили новый блок `except`, чтобы при извлечении страницы управлять исключением `PageNotAnInteger`. Если запрошенная страница не является целым числом, то мы возвращаем первую страницу результатов.

Снова пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=asdf` в своем браузере. Теперь исключение обрабатывается представлением, и первая страница результатов возвращается, как показано ниже на рис. 2.8.

Теперь постраничная разбивка постов блога полностью реализована.

Подробнее о классе `Paginator` можно узнать на странице https://docs.djangoproject.com/en/4.1/ref/paginator/.

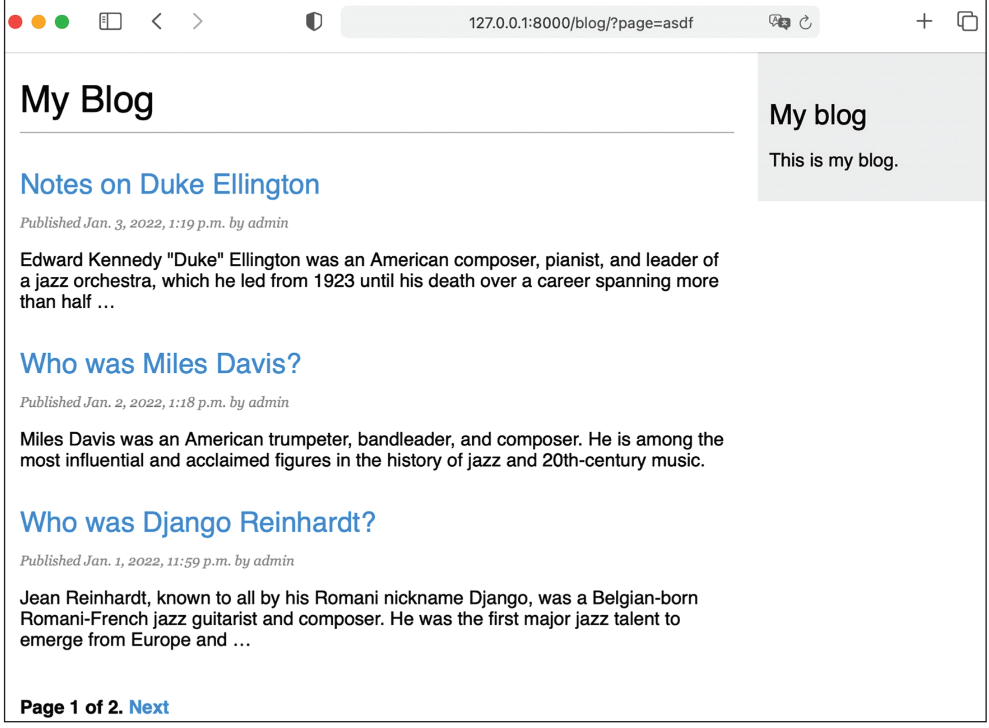

Рис. 2.8. Первая страница результатов

Разработка представлений на основе классов

Мы разработали приложение `blog`, используя представления на основе функций. Такие представления просты и мощны, но Django также позволяет разрабатывать представления с использованием классов.

Представления на основе классов являются альтернативным функциям способом реализации представлений как объектов Python. Поскольку представление – это функция, которая принимает веб-запрос и возвращает веб-ответ, то существует возможность определять представления как методы класса. Django предоставляет базовые классы-представления, которые можно использовать для реализации своих собственных представлений. Все они наследуют от класса `View`, который служит для диспетчеризации HTTP-методов и других распространенных функциональностей.

Зачем использовать представления на основе классов

Представления на основе классов обладают некоторыми преимуществами по сравнению с представлениями на основе функций, которые удобны для конкретных случаев использования. Представления на основе классов позволяют:

- организовывать исходный код, относящийся к HTTP-методам, таким как `GET`, `POST` или `PUT`, в отдельные методы, не используя ветвление по условию;
- использовать множественное наследование, чтобы создавать реиспользуемые классы-представления (также именуемые примесями, примесными классами или миксинами).

Использование представления на основе класса для отображения списка постов

В целях понимания того, как писать представления на основе классов, мы создадим новое представление на основе класса, эквивалентное представлению `post_list`. Мы создадим класс, который будет наследовать от предлагаемого веб-фреймворком Django типового представления `ListView`. Представление `ListView` позволяет перечислять объекты любого типа.

Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный код:

```
from django.views.generic import ListView

class PostListView(ListView):
    """
    Альтернативное представление списка постов
    """
    queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'
```

Представление `PostListView` похоже на разработанное ранее представление `post_list`. Мы имплементировали представление, основанное на классе, которое наследует от класса `ListView`, при этом определив его со следующими атрибутами:

- атрибут `queryset` используется для того, чтобы иметь конкретно-прикладной набор запросов QuerySet, не извлекая все объекты. Вместо определения атрибута `queryset` мы могли бы указать `model=Post`, и Django сформировал бы для нас типовой набор запросов `Post.objects.all()`;
- контекстная переменная `posts` используется для результатов запроса. Если не указано имя контекстного объекта `context_object_name`, то по умолчанию используется переменная `object_list`;
- в атрибуте `paginate_by` задается постраничная разбивка результатов с возвратом трех объектов на страницу;
- конкретно-прикладной шаблон используется для прорисовки страницы шаблоном `template_name`. Если шаблон не задан, то по умолчанию `ListView` будет использовать `blog/post_list.html`.

Теперь отредактируйте файл `urls.py` приложения `blog`, закомментировав предыдущий шаблон URL-адреса `post_list` и добавив новый шаблон URL-адреса, используя класс `PostListView`, как показано ниже:

```
urlpatterns = [
    # представления поста
    # path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
]
```

Для того чтобы постраничная разбивка продолжала работать, необходимо использовать правильный объект страницы, который передается в шаблон. Встроенное в Django типовое представление `ListView` передает запрошенную страницу в переменную с именем `page_obj`. В связи с этим необходимо соответствующим образом отредактировать шаблон `post/list.html`, чтобы вставить разбивщика, используя правильную переменную, как показано ниже:

```
{% 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=page_obj %}
{% endblock %}
```

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и проверьте, чтобы ссылки с постраничной разбивкой работали должным образом. Поведение постранично разбитых ссылок должно быть таким же, как и в предыдущем представлении `post_list`.

Обработка исключений в этом случае немного отличается. Если попытаться загрузить страницу вне диапазона или передать нецелочисленное значение в параметре `page`, то представление вернет HTTP-ответ с кодом состояния, равным `404` (Страница не найдена), как показано ниже.

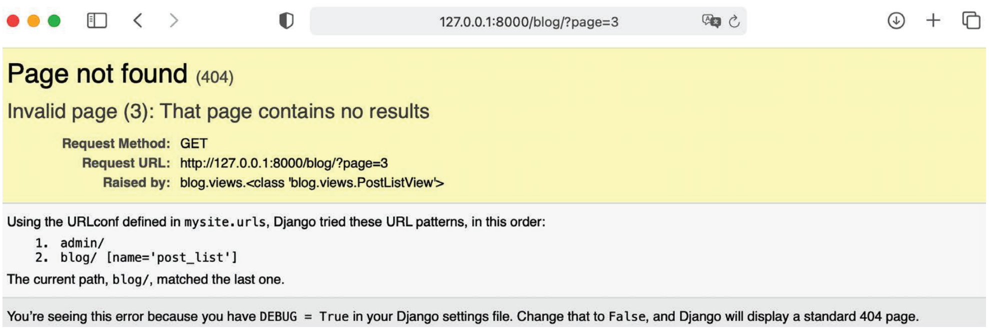

Рис. 2.9. HTTP-ответ с кодом состояния 404 (Страница не найдена)

Обработка исключений, которая возвращает HTTP-ответ с кодом состояния `404`, предусмотрена типовым представлением `ListView`.

Это простой пример того, как писать представления на основе классов. Подробнее о представлениях на основе классов можно узнать в главе 13 «Создание системы управления контентом» и последующих главах.

Введение в представления на основе классов можно почитать на странице https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/.

Рекомендация постов по электронной почте

Теперь мы научимся создавать формы и отправлять электронные письма с помощью Django. Мы предоставим пользователям возможность делиться постами блога с другими, отправляя рекомендуемые посты по электронной почте.

Найдите минутку, чтобы подумать о том, как можно бы использовать представления, URL-адреса и шаблоны для создания этой функциональности, используя то, что вы узнали в предыдущей главе.

Для того чтобы дать пользователям возможность делиться постами по электронной почте, необходимо:

- создать форму, в которой пользователи должны заполнить свое имя, адрес электронной почты, адрес электронной почты получателя и опциональные комментарии;
- создать представление в файле `views.py`, которое обрабатывает опубликованные данные и отправляет электронное письмо;
- добавить шаблон URL-адреса нового представления в файл `urls.py` приложения `blog`;
- создать шаблон отображения формы.

Разработка форм с помощью Django

Давайте начнем с разработки формы, позволяющей делиться постами. Django имеет встроенный фреймворк форм, который позволяет легко создавать формы. Фреймворк форм упрощает определение полей формы, указывает их внешний вид на странице и способы валидации ими входных данных. Встроенный в Django фреймворк форм предлагает гибкий способ прорисовки форм в исходном коде HTML и оперирования данными.

Django поставляется с двумя базовыми классами для разработки форм:

- `Form`: позволяет компоновать стандартные формы путем определения полей и валидаций;
- `ModelForm`: позволяет компоновать формы, привязанные к экземплярам модели. Он предоставляет все функциональности базового класса `Form`, но поля формы можно объявлять явным образом или автоматически генерировать из полей модели. Форму можно использовать для создания либо редактирования экземпляров модели.

Сначала внутри каталога приложения `blog` создайте файл `forms.py` и добавьте в него следующий ниже исходный код:

```
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)
```

Мы определили первую форму Django. Форма `EmailPostForm` наследует от базового класса `Form`. Мы используем различные типы полей, чтобы выполнять валидацию данных в соответствии с ними.

Формы могут находиться в любом месте проекта Django. По традиции их помещают внутри файла `forms.py` в каждом приложении.

Форма содержит следующие ниже поля:

- `name`: экземпляр класса `CharField` с максимальной длиной 25 символов, который будет использоваться для имени человека, отправляющего пост;
- `email`: экземпляр класса `EmailField`. Здесь используется адрес электронной почты человека, отправившего рекомендуемый пост;
- `to`: экземпляр класса `EmailField`. Здесь используется адрес электронной почты получателя, который будет получать электронное письмо с рекомендуемым постом;
- `comments`: экземпляр класса `CharField`. Он используется для комментариев, которые будут вставляться в электронное письмо с рекомендуемым постом. Это поле сделано опциональным путем установки `required` равным значению `False`, при этом был задан конкретно-прикладной виджет прорисовки поля.

У каждого типа поля есть заранее заданный виджет, который определяет то, как поле прорисовывается в исходном коде HTML. Поле `name` является экземпляром класса `CharField`. Поле этого типа прорисовывается как HTML-элемент `<input type="text">`. Заранее заданный виджет можно переопределять посредством атрибута `widget`. В поле `comments` используется виджет `Textarea`, чтобы отображать его как HTML-элемент `<textarea>` вместо используемого по умолчанию элемента `<input>`.

Валидация полей также зависит от типа полей. Например, поля `email` и `to` являются полями типа `EmailField`. Для обоих полей требуется валидный адрес электронной почты; в противном случае валидация поля вызовет исключение `forms.ValidationError`, и форма не пройдет валидацию. При валидации полей формы также принимаются во внимание и другие параметры, такие как поле `name`, имеющее максимальную длину 25, или поле `comments`, являющееся опциональным.

Это лишь некоторые типы полей, которые Django предоставляет для форм. Список всех имеющихся типов полей находится на странице https://docs.djangoproject.com/en/4.1/ref/forms/fields/.

Работа с формами в представлениях

Мы определили форму для рекомендации постов по электронной почте. Теперь требуется представление, чтобы создавать экземпляр формы и работать с передачей формы на обработку.

Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный код:

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # Извлечь пост по идентификатору id
    post = get_object_or_404(Post,
                             id=post_id,
                             status=Post.Status.PUBLISHED)
    if request.method == 'POST':
        # Форма была передана на обработку
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Поля формы успешно прошли валидацию
            cd = form.cleaned_data
            # ... отправить электронное письмо
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form})
```

Мы определили представление `post_share`, которое в качестве параметров принимает объект `request` и переменную `post_id`. Мы используем функцию сокращенного доступа `get_object_or_404()`, чтобы извлечь опубликованный пост по его `id`.

Одно и то же представление используется как для отображения изначальной формы на странице, так и для обработки представленных для валидации данных. HTTP-метод `request` позволяет различать случаи, когда форма передается на обработку. Запрос `GET` будет указывать на то, что пользователю должна быть отображена пустая форма, а запрос `POST` – на то, что форма передается на обработку. Булево выражение `request.method == 'POST'` используется для того, чтобы проводить различие между этими двумя сценариями.

Ниже описывается процесс отображения формы на странице и работы с передачей формы на обработку.

1. Когда страница загружается в первый раз, представление получает запрос `GET`. В этом случае создается новый экземпляр класса `EmailPostForm`, который сохраняется в переменной `form`. Указанный экземпляр формы будет использоваться для отображения пустой формы в шаблоне:

```
form = EmailPostForm()
```

2. Когда пользователь заполняет форму и передает ее методом `POST` на обработку, создается экземпляр формы с использованием переданных данных, содержащихся в `request.POST`:

```
if request.method == 'POST':
    # Форма была передана на обработку
    form = EmailPostForm(request.POST)
```

3. После этого переданные данные валидируются методом `is_valid()` формы. Указанный метод проверяет допустимость введенных в форму данных и возвращает значение `True`, если все поля содержат валидные данные. Если какое-либо поле содержит невалидные данные, то `is_valid()` возвращает значение `False`. Список ошибок валидации можно получить посредством `form.errors`.
4. Если форма невалидна, то форма снова прорисовывается в шаблоне, включая переданные данные. Ошибки валидации будут отображены в шаблоне.
5. Если форма валидна, то валидированные данные извлекаются посредством `form.cleaned_data`. Указанный атрибут представляет собой словарь полей формы и их значений.

Если данные формы не проходят валидацию, то `cleaned_data` будет содержать только валидные поля.

Мы реализовали представление отображения формы на странице и передачи формы на обработку. Теперь мы научимся отправлять электронные письма с помощью Django и затем добавим эту функциональность в представление `post_share`.

Отправка электронных писем с помощью Django

Отправка электронных писем в Django очень проста. Для того чтобы отправлять электронные письма с помощью Django, необходимо иметь локальный **SMTP**-сервер[1] (работающий по простому протоколу передачи почты) либо обращаться к внешнему SMTP-серверу, например к своему поставщику услуг электронной почты.

Следующие ниже настроечные параметры позволяют определять конфигурацию SMTP, чтобы отправлять электронные письма с помощью Django:

- `EMAIL_HOST`: хост SMTP-сервера; по умолчанию используется `localhost`;
- `EMAIL_PORT`: SMTP-порт; по умолчанию равен `25`;
- `EMAIL_HOST_USER`: пользовательское имя для SMTP-сервера;
- `EMAIL_HOST_PASSWORD`: пароль для SMTP-сервера;
- `EMAIL_USE_TLS`: следует ли использовать защищенное соединение транспортного слоя (**TLS**)[2];
- `EMAIL_USE_SSL`: следует ли использовать неявное защищенное соединение TLS.

В этом примере мы будем использовать SMTP-сервер Google со стандартной учетной записью Gmail.

---

[1] Англ. Simple Mail Transfer Protocol. – *Прим. перев.*
[2] Англ. Transport Layer Security. – *Прим. перев.*

Если у вас есть учетная запись Gmail, то отредактируйте файл settings.py проекта, добавив следующий ниже исходный код:

```
# Конфигурация сервера электронной почты
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = ''
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Замените your_account@gmail.com своей реальной учетной записью Gmail. Если у вас нет учетной записи Gmail, то можете использовать конфигурацию SMTP-сервера своего поставщика услуг электронной почты.

Вместо Gmail также можно использовать профессиональный масштабируемый почтовый сервис, который позволяет отправлять электронные письма по протоколу SMTP, используя ваш собственный домен. Например, SendGrid (https://sendgrid.com/) или простой почтовый сервис Amazon (https://aws.amazon.com/ses/). Оба сервиса потребуют подтверждения домена и учетных записей электронной почты отправителя и предоставят учетные данные SMTP для отправки электронных писем. Приложения Django django-sengrid и django-ses упрощают задачу добавления сервисов SendGrid или Amazon SES в свой проект. Инструкции по установке django-sengrid находятся на странице https://github.com/sklarsa/django-sendgrid-v5, инструкции по установке django-ses расположены на странице https://github.com/django-ses/django-ses.

Если вы не можете использовать SMTP-сервер, то можно сообщить Django, что нужно писать электронные письма в консоль, добавив в файл settings.py следующий ниже настроечный параметр:

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

Используя этот параметр, Django будет выводить все электронные письма в оболочку, не отправляя их. Это бывает очень удобно при тестировании своего приложения без SMTP-сервера.

Завершая конфигурирование Gmail, необходимо ввести пароль для SMTP-сервера. Поскольку Google использует двухэтапный процесс верификации и дополнительные меры безопасности, вы не сможете использовать пароль своей учетной записи Google напрямую. Вместо этого Google позволяет создавать конкретно-прикладные пароли в вашем аккаунте. Пароль приложения – это 16-значный код доступа, который дает менее защищенному приложению или устройству разрешение на доступ к вашей учетной записи Google.

Пройдите по URL-адресу https://myaccount.google.com/ в своем браузере. В левом меню выберите пункт **Security** (Безопасность). Вы увидите следующий ниже экран:

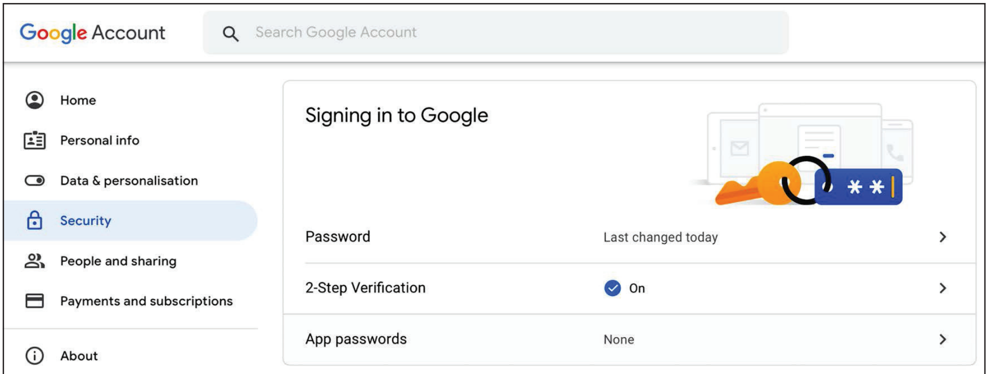

Рис. 2.10. Вход в Google для учетных записей Google

В разделе **Signing in to Google** (Вход в Google) кликните по **App passwords** (Пароли приложений). Если вы не видите пароли приложений, то, возможно, для вашей учетной записи не настроена двухэтапная верификация, ваша учетная запись является учетной записью организации, а не стандартной учетной записью Gmail, либы вы задействовали расширенную защиту Google. Проверьте, чтобы использовалась стандартная учетная запись Gmail и была активирована двухэтапная верификация своей учетной записи Google. Более подробная информация находится на странице https://support.google.com/accounts/answer/185833.

При нажатии на **App passwords** вы увидите следующий ниже экран:

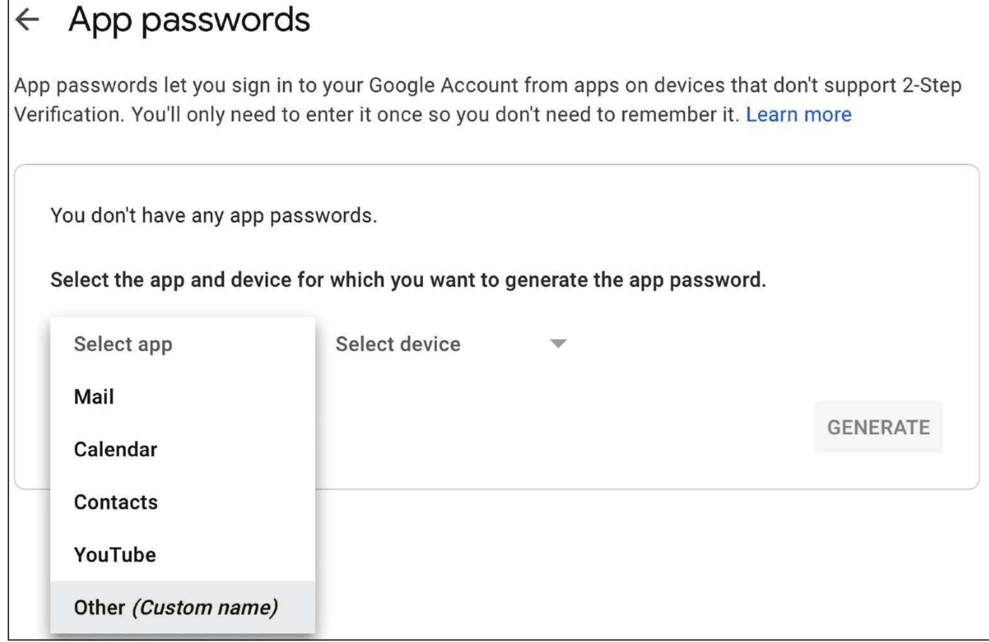

Рис. 2.11. Форма для генерирования нового пароля приложения Google

В ниспадающем списке **Select app** (Выбрать приложение) выберите **Other** (Другое).

Затем введите имя `Blog` и кликните по кнопке **GENERATE** (Сгенерировать), как показано ниже:

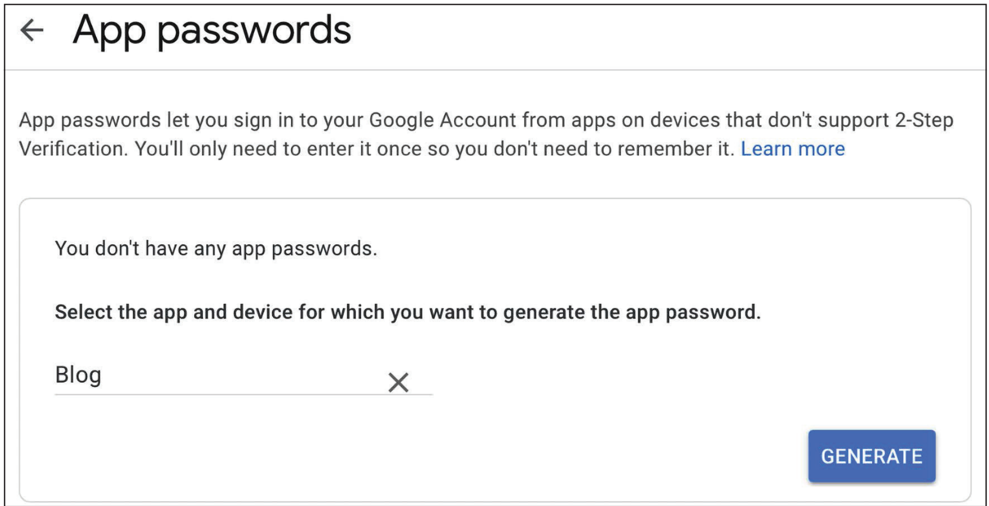

Рис. 2.12. Форма для генерирования нового пароля приложения Google

Новый пароль будет сгенерирован и выведен на страницу, как показано ниже:

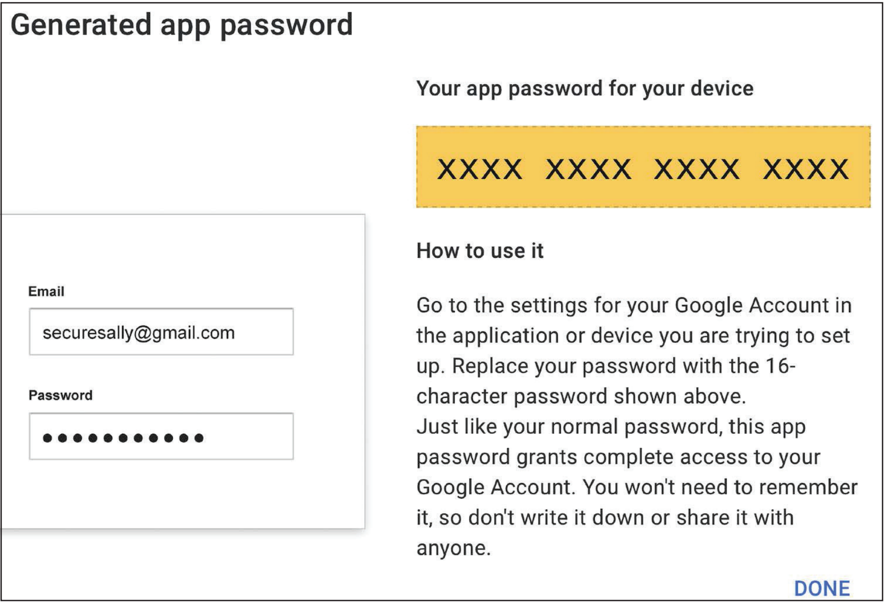

Рис. 2.13. Сгенерированный пароль приложения Google

Скопируйте сгенерированный пароль приложения.

Отредактируйте файл `settings.py` проекта, добавив пароль приложения в настроечный параметр `EMAIL_HOST_PASSWORD`, как показано ниже:

```
# Конфигурация сервера электронной почты
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'xxxxxxxxxxxxxxxx'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Откройте оболочку Python, выполнив следующую ниже команду в командной строке системной оболочки:

```
python manage.py shell
```

Исполните следующий ниже исходный код в оболочке Python:

```
>>> 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)
```

Функция `send_mail()` принимает тему, сообщение, отправителя и список получателей в качестве требуемых аргументов. Устанавливая опциональный аргумент `fail_silently=False`, мы сообщаем ей, что если электронное письмо невозможно отправить, нужно вызывать исключение. Если результат, который вы видите, равен 1, значит, ваше электронное письмо было успешно отправлено.

Проверьте свой почтовый ящик. Вы должны были получить электронное письмо:

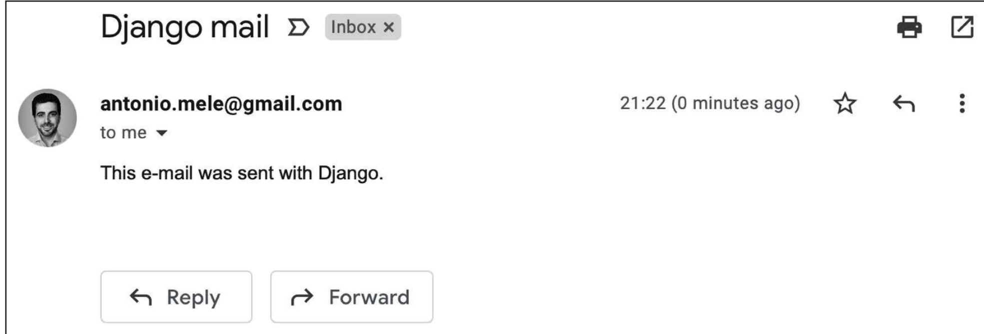

Рис. 2.14. Отправленное тестовое электронное письмо отображается в Gmail

Вы только что отправили свое первое электронное письмо с помощью Django! Более подробная информация об отправке электронных писем с помощью Django находится на странице https://docs.djangoproject.com/en/4.1/topics/email/.

Давайте добавим эту функциональность в представление post_share.

Отправка электронных писем в представлениях

Отредактируйте представление post_share в файле views.py приложения blog, как показано ниже:

```
from django.core.mail import send_mail

def post_share(request, post_id):
    # Извлечь пост по его идентификатору id
    post = get_object_or_404(Post,
                             id=post_id,
                             status=Post.Status.PUBLISHED)
    sent = False

    if request.method == 'POST':
        # Форма была передана на обработку
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Поля формы успешно прошли валидацию
            cd = form.cleaned_data
            post_url = request.build_absolute_uri(
                post.get_absolute_url())
            subject = f"{cd['name']} recommends you read " \
                      f"{post.title}"
            message = f"Read {post.title} at {post_url}\n\n" \
                      f"{cd['name']}\'s comments: {cd['comments']}"
            send_mail(subject, message, 'your_account@gmail.com',
                      [cd['to']])
            sent = True
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form,
                                                    'sent': sent})
```

Если вы используете SMTP-сервер, а не почтовый бэкенд[1] console.EmailBackend, то замените your_account@gmail.com своей реальной учетной записью электронной почты.

---

[1] Син. серверная часть веб-приложения. – *Прим. перев.*

В приведенном выше исходном коде мы объявили переменную `sent` с изначальным значением `False`. Мы задаем этой переменной значение `True` после отправки электронного письма. Позже мы будем использовать переменную `sent` в шаблоне отображения сообщения об успехе при успешной передаче формы.

Поскольку ссылка на пост должна вставляться в электронное письмо, мы получаем абсолютный путь к посту, используя его метод `get_absolute_url()`. Мы используем этот путь на входе в метод `request.build_absolute_uri()`, чтобы сформировать полный URL-адрес, включая HTTP-схему и хост-имя (hostname)[1].

Мы создаем тему и текст сообщения электронного письма, используя очищенные данные валидированной формы. Наконец, мы отправляем электронное письмо на адрес электронной почты, указанный в поле `to` (Кому) формы.

Теперь, когда представление `post_share` завершено, для него необходимо добавить новый шаблон URL-адреса.

Откройте файл `urls.py` приложения `blog` и добавьте шаблон URL-адреса `post_share`, как показано ниже:

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

app_name = 'blog'

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

Прорисовка форм в шаблонах

После того как была создана форма, запрограммировано представление и добавлен шаблон URL-адреса, не хватает только одного – шаблона представления.

Внутри каталога `blog/templates/blog/post/` создайте новый файл и назовите его `share.html`.

Добавьте следующий ниже исходный код в новый шаблон `share.html`:

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

[1] Син. сетевое имя, имя узла. – *Прим. перев.*

```
{% 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 %}
```

Это шаблон, который используется для отображения формы, служащей для того, чтобы делиться постом по электронной почте, и для отображения успешного сообщения после отправки электронного письма. Различие между обоими случаями проводится с помощью тега `{% if sent %}`.

Для того чтобы отобразить форму, мы определили HTML-элемент `form`, указав, что форма должна быть передана методом `POST`:

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

Экземпляр формы вставлен с помощью тега `{{ form.as_p }}`. При этом веб-фреймворку Django сообщается, что нужно прорисовывать поля формы, используя абзацные HTML-элементы `<p>` с применением метода `as_p`. Кроме того, форму можно было бы прорисовывать в виде неупорядоченного списка с использованием метода `as_ul` или в виде HTML-таблицы с применением метода `as_table`. Еще одним вариантом является прорисовка каждого поля путем прокручивания полей формы в цикле, как в следующем ниже примере:

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

Здесь добавлен шаблонный тег `{% csrf_token %}`. Указанный тег вводит скрытое поле с автоматически сгенерированным токеном во избежание атак

по подделке межсайтовых запросов (**CSRF**)[1]. Такие атаки заключаются в том, что вредоносный веб-сайт или программа выполняют нежелательные для пользователя действия на сайте. Более подробная информация о подделке межсайтовых запросов находится на странице https://owasp.org/www-community/attacks/csrf.

Шаблонный тег `{% csrf_token %}` генерирует скрытое поле, которое прорисовывается следующим образом:

```
<input type='hidden' name='csrfmiddlewaretoken'
value='26JjKo2lcEtYkGoV9z4XmJIEHLXN5LDR' />
```

По умолчанию Django проверяет наличие токена CSRF во всех запросах методом POST. Тег csrf_token следует вставлять во все формы, передаваемые на обработку методом POST.

Отредактируйте шаблон `blog/post/detail.html`, придав ему следующий вид:

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

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

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

Здесь была добавлена ссылка на URL-адрес `post_share`. URL-адрес формируется динамически с помощью предоставляемого веб-фреймворком Django шаблонного тега `{% url %}`. При этом используются именное пространство `blog` и URL-адрес `post_share`. Для того чтобы сформировать URL-адрес, `id` поста передается в качестве параметра.

Откройте приглашение командной оболочки и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

[1] Англ. Cross-Site Request Forgery. – *Прим. перев.*

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и кликните по заголовку любого поста, чтобы просмотреть страницу детальной информации о посте.

Под телом поста вы должны увидеть ссылку, которую вы только что добавили, как показано на рис. 2.15.

Рис. 2.15. Страница детальной информации о посте, включая ссылку, чтобы поделиться постом

Кликните по **Share this post** (Поделиться этим постом), и вы должны увидеть страницу, включая форму, позволяющую делиться этим постом по электронной почте, как показано ниже:

Рис. 2.16. Страница, позволяющая делиться постом по электронной почте

Стили CSS формы включены в пример исходного кода и находятся в файле `static/css/blog.css`. При нажатии кнопки **SEND E-MAIL** (Отправить электронное письмо) форма передается на обработку и затем валидируется. Если

все поля содержат валидные данные, то вы получите сообщение об успехе, как показано ниже:

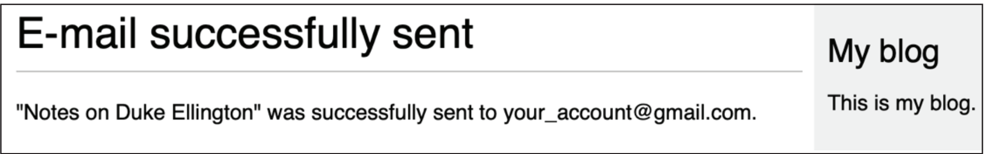

Рис. 2.17. Сообщение об успехе для поста, отправленного по электронной почте

Отправьте пост на свой собственный адрес электронной почты и проверьте свой почтовый ящик. Полученное вами электронное письмо должно выглядеть следующим образом:

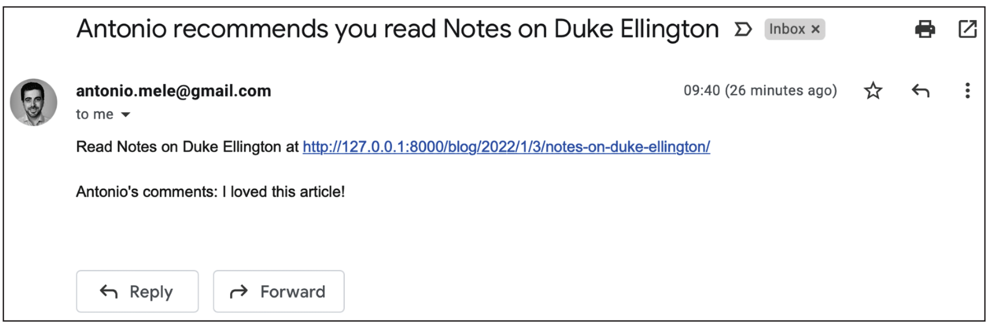

Рис. 2.18. Отправленное тестовое электронное письмо отображается в Gmail

Если передать форму на обработку с невалидными данными, то форма будет прорисована снова, включая все ошибки валидации (рис. 2.19).

Большинство современных браузеров не позволят передавать форму на обработку с пустыми или ошибочными полями. Это вызвано тем, что перед передачей формы на обработку браузер проверяет поля на основе их атрибутов. В этом случае форма не будет передана на обработку, и браузер отобразит сообщение об ошибке у неправильных полей. Для того чтобы протестировать валидацию формы Django с использованием современного браузера, можно пропустить валидацию формы браузером, добавив атрибут `novalidate` в элемент HTML `<form>`. Например, `<form method="post" novalidate>`. Этот атрибут можно добавлять, чтобы запрещать браузеру валидировать поля и тестировать свою собственную валидацию формы. После завершения тестирования удалите атрибут `novalidate`, чтобы вернуть валидацию формы браузером.

Теперь функциональность, позволяющая делиться постами по электронной почте, завершена. Более подробная информация о работе с формами находится на странице https://docs.djangoproject.com/en/4.1/topics/forms/.

Share "Notes on Duke Ellington" by e-mail

My blog

This is my blog.

Name:

Antonio

• Enter a valid email address.

Email:

Invalid

• This field is required.

To:

Comments:

SEND E-MAIL

Рис. 2.19. Форма для отправки поста, отображающая ошибки недопустимости данных

Создание системы комментариев

Мы продолжим работу над расширением приложения для ведения блога, разработав систему комментариев, которая позволит пользователям комментировать посты. Для того чтобы разработать такую систему, понадобится:

- модель комментария, чтобы хранить комментарии пользователей к постам;
- форма, которая позволяет пользователям передавать комментарии на обработку и управляет валидацией данных;
- представление, которое обрабатывает форму и сохраняет новый комментарий в базе данных;
- список комментариев и форма, чтобы добавлять новый комментарий, который может быть вставлен в шаблон детальной информации о посте.

Разработка модели комментария

Давайте начнем с разработки модели для хранения комментариев пользователей к постам.

Откройте файл `models.py` приложения `blog` и добавьте в него следующий ниже исходный код:

```
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}'
```

Это модель `Comment`. Поле `ForeignKey` было добавлено для того, чтобы связать каждый комментарий с одним постом. Указанная взаимосвязь многие-к-одному определена в модели `Comment`, потому что каждый комментарий будет делаться к одному посту, и каждый пост может содержать несколько комментариев.

Атрибут `related_name` позволяет назначать имя атрибуту, который используется для связи от ассоциированного объекта назад к нему. Пост комментарного объекта можно извлекать посредством `comment.post` и все комментарии, ассоциированные с объектом-постом, – посредством `post.comments.all()`. Если атрибут `related_name` не определен, то Django будет использовать имя модели в нижнем регистре, за которым следует `_set` (то есть `comment_set`), чтобы именовать взаимосвязь ассоциированного объекта с объектом модели, в которой эта взаимосвязь была определена.

Подробнее о взаимосвязях многие-к-одному можно узнать на странице https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/.

Мы определили булево поле `active`, чтобы управлять статусом комментариев. Данное поле позволит деактивировать неуместные комментарии вручную с помощью сайта администрирования. Мы используем параметр `default=True`, чтобы указать, что по умолчанию все комментарии активны.

Мы определили поле `created`, чтобы хранить дату и время создания комментария. Используя `auto_now_add`, дата будет сохраняться автоматически при создании объекта. В `Meta`-класс модели был добавлен атрибут `ordering = ['created']`, чтобы по умолчанию сортировать комментарии в хронологическом порядке и индексировать поля `created` в возрастающем порядке.

В результате этого будет повышена производительность операций поиска в базе данных и упорядочивания результатов с использованием поля `created`.

Разработанная модель `Comment` не синхронизирована с базой данных, и поэтому необходимо сгенерировать новую миграцию в базе данных, чтобы создать соответствующую таблицу базы данных.

Выполните следующую ниже команду из командной оболочки:

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

Вы должны увидеть следующий ниже результат:

```
Migrations for 'blog':
  blog/migrations/0003_comment.py
    - Create model Comment
```

Django сгенерировал файл `0003_comment.py` внутри каталога `migrations/` приложения `blog`. Теперь необходимо создать соответствующую схему базы данных и применить изменения к базе данных.

Выполните следующую ниже команду, чтобы применить существующие миграции:

```
python manage.py migrate
```

Вы получите результат, который содержит следующую ниже строку:

```
Applying blog.0003_comment... OK
```

Миграция была применена, и в базе данных была создана таблица `blog_comment`.

Добавление комментариев на сайт администрирования

Далее мы добавим новую модель на сайт администрирования, чтобы управлять комментариями через простой интерфейс.

Откройте файл `admin.py` приложения `blog`, импортируйте модель `Comment` и добавьте следующее:

```
from .models import Post, Comment

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

Откройте командную оболочку и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/` в своем браузере. Вы должны увидеть, что новая модель была вставлена в раздел **BLOG**, как показано на рис. 2.20.

Рис. 2.20. Модели приложения для ведения блога на индексной странице сайта администрирования

Теперь модель зарегистрирована на сайте администрирования.

В строке **Comments** (Комментарии) кликните по **Add** (Добавить). Вы увидите форму для добавления нового комментария:

Рис. 2.21. Форма для добавления нового комментария на сайте администрирования

Теперь появилась возможность управлять экземплярами комментариев с помощью сайта администрирования.

Создание форм из моделей

Далее необходимо скомпоновать форму, позволяющую пользователям комментировать посты блога. Напомним, что в Django есть два базовых класса, которые можно использовать для создания форм: `Form` и `ModelForm`. Мы использовали класс `Form`, чтобы предоставлять пользователям возможность делиться постами по электронной почте. Теперь мы будем использовать `ModelForm`, чтобы воспользоваться преимуществами существующей модели `Comment` и компоновать для нее форму динамически.

Отредактируйте файл `forms.py` приложения `blog`, добавив следующие ниже строки:

```
from .models import Comment

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

Для того чтобы создать форму из модели, надо в `Meta`-классе формы просто указать модель, для которой следует компоновать форму. Django проведет интроспекцию модели и динамически скомпонует соответствующую форму.

Каждому типу поля модели соответствует заранее заданный тип поля формы. Атрибуты полей модели учитываются при валидации формы. По умолчанию Django создает поле формы для каждого содержащегося в модели поля. Однако, используя атрибут `fields`, можно сообщать поля, которые следует включать в форму, либо, используя атрибут `exclude`, сообщать поля, которые следует исключать, задавая поля в явном виде. В форме `CommentForm` мы включили поля `name`, `email` и `body` в явном виде. Это единственные поля, которые будут включены в форму.

Более подробная информация о создании форм из моделей находится на странице https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/.

Оперирование формами ModelForm в представлениях

Для того чтобы делиться постами по электронной почте, мы использовали одно и то же представление, которое служило как для отображения формы, так и для управления ее передачей на обработку. Мы использовали HTTP-метод, чтобы проводить различие между обоими случаями, `GET`, чтобы ото-

бражать форму на странице, и `POST`, чтобы передавать ее на обработку. В этом случае мы добавим комментарную форму на страницу детальной информации о посте и разработаем отдельное представление, которое посвящено передаче формы на обработку. Новое обрабатывающее форму представление позволит пользователю возвращаться к представлению детальной информации о посте, после того как комментарий будет сохранен в базе данных.

Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный код:

```
from django.shortcuts import render, get_object_or_404, redirect
from .models import Post, Comment
from django.core.paginator import Paginator, EmptyPage,\
                                  PageNotAnInteger
from django.views.generic import ListView
from .forms import EmailPostForm, CommentForm
from django.core.mail import send_mail
from django.views.decorators.http import require_POST

# ...

@require_POST
def post_comment(request, post_id):
    post = get_object_or_404(Post,
                             id=post_id,
                             status=Post.Status.PUBLISHED)
    comment = None
    # Комментарий был отправлен
    form = CommentForm(data=request.POST)
    if form.is_valid():
        # Создать объект класса Comment, не сохраняя его в базе данных
        comment = form.save(commit=False)
        # Назначить пост комментарию
        comment.post = post
        # Сохранить комментарий в базе данных
        comment.save()
    return render(request, 'blog/post/comment.html',
                           {'post': post,
                            'form': form,
                            'comment': comment})
```

Мы определили представление `post_comment`, которое принимает объект `request` и переменную `post_id` в качестве параметров. Мы будем использовать это представление, чтобы управлять передачей поста на обработку. Мы ожидаем, что форма будет передаваться с использованием HTTP-метода `POST`. Мы используем предоставляемый веб-фреймворком Django декоратор `require_POST`, чтобы разрешить запросы методом `POST` только для этого представления. Django позволяет ограничивать разрешенные для представлений

HTTP-методы. Если пытаться обращаться к представлению посредством любого другого HTTP-метода, то Django будет выдавать ошибку HTTP 405 (Метод не разрешен).

В этом представлении реализованы следующие ниже действия.

1. По id поста извлекается опубликованный пост, используя функцию сокращенного доступа get_object_or_404().
2. Определяется переменная comment с изначальным значением None. Указанная переменная будет использоваться для хранения комментарного объекта при его создании.
3. Создается экземпляр формы, используя переданные на обработку POST-данные, и проводится их валидация методом is_valid(). Если форма невалидна, то шаблон прорисовывается с ошибками валидации.
4. Если форма валидна, то создается новый объект Comment, вызывая метод save() формы, и назначается переменной new_comment, как показано ниже:

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

5. Метод save() создает экземпляр модели, к которой форма привязана, и сохраняет его в базе данных. Если вызывать его, используя commit=False, то экземпляр модели создается, но не сохраняется в базе данных. Такой подход позволяет видоизменять объект перед его окончательным сохранением.

Метод save() доступен для ModelForm, но не для экземпляров класса Form, поскольку они не привязаны ни к одной модели.

6. Пост назначается созданному комментарию:

```
comment.post = post
```

7. Новый комментарий создается в базе данных путем вызова его метода save():

```
comment.save()
```

8. Прорисовывается шаблон blog/post/comment.html, передавая объекты post, form и comment в контекст шаблона. Этот шаблон еще не существует; мы создадим его позже.

Давайте создадим шаблон URL-адреса этого представления.

Отредактируйте файл urls.py приложения blog, добавив следующий ниже шаблон URL-адреса:

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

app_name = 'blog'

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

Мы реализовали представление, чтобы управлять передачей комментариев на обработку и соответствующими им URL-адресами. Давайте создадим необходимые шаблоны.

Создание шаблонов комментарной формы

Мы создадим шаблон комментарной формы, которая будет использоваться в двух местах:

- в шаблоне детальной информации о посте, ассоциированном с представлением `post_detail`, чтобы пользователи могли публиковать комментарии;
- в шаблоне комментария к посту, ассоциированном с представлением `post_comment`, чтобы отображать форму снова, если в форме есть какие-либо ошибки.

Мы создадим шаблон формы и будем использовать шаблонный тег `{% include %}`, чтобы вставлять его в два других шаблона.

Внутри каталога `templates/blog/post/` создайте новый каталог `includes/`. В этот каталог добавьте новый файл и назовите его `comment_form.html`.

Файловая структура должна выглядеть следующим образом:

```
templates/
  blog/
    post/
      includes/
        comment_form.html
      detail.html
```

```
      list.html
      share.html
```

Отредактируйте новый шаблон `blog/post/includes/comment_form.html`, добавив следующий ниже исходный код:

```
<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>
```

В указанном шаблоне мы динамически формируем URL-адрес `action` HTML-элемента `<form>`, используя шаблонный тег `{% url %}`. Мы формируем URL-адрес представления `post_comment`, которое будет обрабатывать форму. Мы отображаем форму, прорисованную абзацами HTML, и вставляем тег `{% csrf_token %}`, чтобы защититься от CSRF, поскольку данная форма будет передаваться на обработку методом `POST`.

Внутри каталога `templates/blog/post/` создайте новый файл приложения `blog` и назовите его `comment.html`.

Теперь файловая структура должна выглядеть следующим образом:

```
templates/
  blog/
    post/
      includes/
        comment_form.html
      comment.html
      detail.html
      list.html
      share.html
```

Отредактируйте новый шаблон `blog/post/comment.html`, добавив следующий ниже исходный код:

```
{% 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 %}
```

Это шаблон представления комментариев к посту. В данном представлении мы ожидаем, что форма будет передаваться на обработку методом `POST`. Шаблон охватывает два разных сценария:

- если переданные данные формы валидны, то переменная `comment` будет содержать созданный объект `comment`, и на страницу будет выведено сообщение об успехе;
- если переданные данные формы невалидны, то переменной `comment` будет назначено значение `None`. В этом случае мы отобразим комментарную форму. Для вставки созданного ранее шаблона `comment_form.html` используется шаблонный тег `{% include %}`.

Добавление комментариев в представление детальной информации о посте

Откройте файл `views.py` приложения `blog` и отредактируйте представление `post_detail`, как показано ниже:

```
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)
    # Список активных комментариев к этому посту
    comments = post.comments.filter(active=True)
    # Форма для комментирования пользователями
    form = CommentForm()
    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                   'comments': comments,
                   'form': form})
```

Давайте рассмотрим исходный код, который мы добавили в представление `post_detail`:

- мы добавили набор запросов QuerySet, чтобы извлекать все активные комментарии к посту, как показано ниже:

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

- этот набор запросов сформирован с использованием объекта post. Вместо того чтобы формировать набор запросов для комментарной модели напрямую, мы используем объект post, чтобы извлекать связанные объекты Comment. Мы применяем менеджер comments для ранее определенных в модели Comment связанных с Comment объектов, используя атрибут related_name поля ForeignKey в модели Post;
- мы также создали экземпляр формы для комментария посредством инструкции form = CommentForm().

Добавление комментариев в шаблон детальной информации о посте

Далее необходимо отредактировать шаблон blog/post/detail.html, чтобы реализовать следующее:

- показывать общее число комментариев к посту;
- показывать список комментариев;
- показывать форму для добавления пользователями новых комментариев.

Мы начнем с добавления общего числа комментариев к посту.

Отредактируйте шаблон blog/post/detail.html, внеся в него изменения, как показано ниже:

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

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

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
{% endblock %}
```

В указанном шаблоне мы используем Django ORM-преобразователь, применяя набор запросов `comments.count()`. Обратите внимание, что на языке шаблонов Django для вызова методов круглые скобки не используются. Тег `{% with %}` позволяет присваивать значение новой переменной, которая будет доступна в шаблоне до тех пор, пока не появится тег `{% endwith %}`.

Шаблонный тег `{% with %}` полезен тем, что он позволяет избегать многократного обращения к базе данных или к дорогостоящим методам.

Мы используем шаблонный фильтр `pluralize`, чтобы отображать суффикс множественного числа для слова *comment*, в зависимости от значения `total_comments`. Шаблонные фильтры на входе принимают значение переменной, к которой они применяются, и на выходе возвращают вычисленное значение. Подробнее о шаблонных фильтрах мы узнаем в главе 3 «Расширение приложения для ведения блога».

Шаблонный фильтр `pluralize` возвращает строковый литерал с буквой «s», если значение отличается от 1. Приведенный выше текст будет прорисовываться как *0 comments*, *1 comment* или *N comments*, в зависимости от числа активных комментариев к посту.

Теперь давайте добавим список активных комментариев в шаблон детальной информации о посте.

Отредактируйте шаблон `blog/post/detail.html`, внеся в него следующие ниже изменения:

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

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

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
```

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

Мы добавили шаблонный тег `{% for %}`, чтобы прокручивать комментарии к посту в цикле. Если список комментариев пуст, то выводится сообщение, информирующее пользователей о том, что комментариев к этому посту нет. Комментарии прокручиваются в цикле посредством переменной `{{ forloop.counter }}`, которая обновляет счетчик цикла на каждой итерации. По каждому посту мы показываем имя пользователя, который его опубликовал, дату и текст комментария.

Наконец, давайте добавим форму комментария в шаблон.

Отредактируйте шаблон `blog/post/detail.html`, вставив шаблон комментарной формы, как показано ниже:

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

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

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
```

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

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и кликните по заголовку поста, чтобы взглянуть на страницу подробного описания поста. Вы увидите что-то вроде рис. 2.22:

Рис. 2.22. Страница детальной информации о посте, содержащая форму для добавления комментария

Заполните форму валидными данными и кликните по **Add comment** (Добавить комментарий). Вы должны увидеть следующую ниже страницу:

Рис. 2.23. Страница успешного добавления комментария

Кликните по ссылке **Back to the post** (Вернуться к посту). Вы должны быть перенаправлены обратно на страницу детальной информации о посте и увидеть комментарий, который вы только что добавили, как показано ниже:

Рис. 2.24. Страница детальной информации о посте, включая комментарий

Добавьте еще один комментарий в этот пост. Комментарии должны располагаться под содержимым поста в хронологическом порядке, как показано ниже:

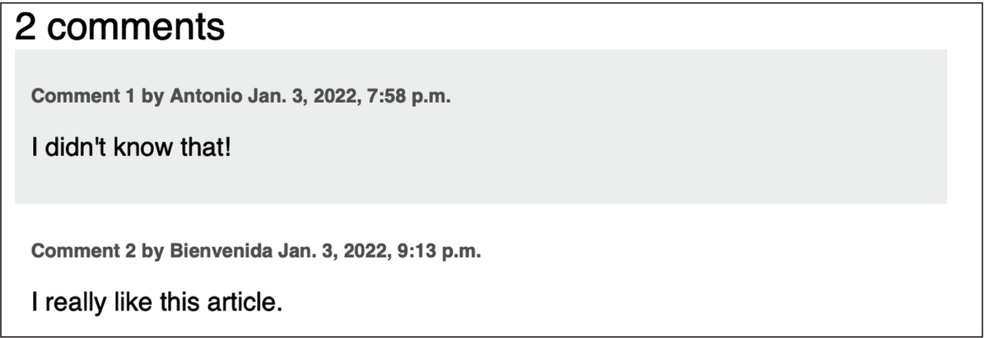

Рис. 2.25. Список комментариев на странице детальной информации о посте

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/blog/comment/` в своем браузере. Вы увидите страницу администрирования со списком созданных вами комментариев. Например, как на рис. 2.26.

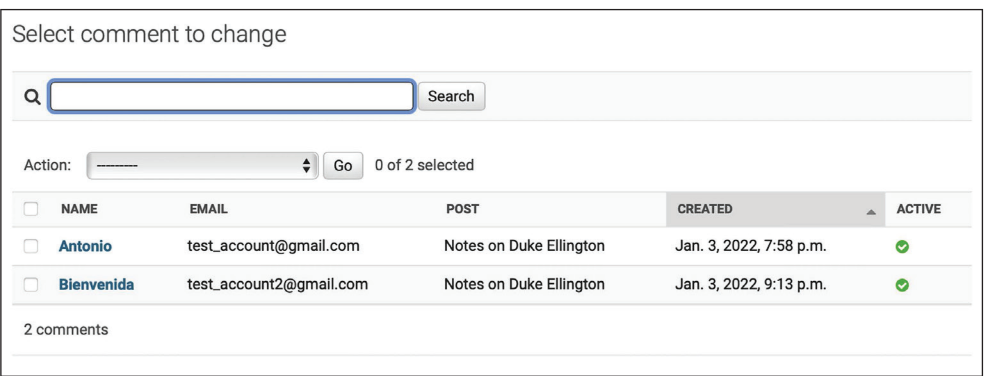

Рис. 2.26. Список комментариев на сайте администрирования

Кликните по заголовку одного из постов, чтобы его отредактировать. Снимите флажок **Active** (Активен), как показано ниже, и кликните по кнопке **Save** (Сохранить):

Рис. 2.27. Редактирование комментария на сайте администрирования

Вы будете перенаправлены на список комментариев. В столбце **Active** отобразится неактивный значок комментария, как показано на рис. 2.28:

Рис. 2.28. Активные/неактивные комментарии на сайте администрирования

Если вернуться к представлению детальной информации о посте, то можно заметить, что неактивный комментарий больше не отображается и не учитывается при подсчете общего числа активных комментариев к посту:

1 comment

Comment 1 by Bienvenida Jan. 3, 2022, 9:13 p.m.

I really like this article.

Рис. 2.29. Один активный комментарий, отображаемый на странице детальной информации о посте

Благодаря полю **Active** можно деактивировать неуместные комментарии и не показывать их в своих постах.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02.
- Функции-утилиты для URL-адресов: https://docs.djangoproject.com/en/4.1/ref/urlresolvers/.
- Конверторы путей URL-адресов: https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters.
- Встроенный в Django класс постраничной разбивки: https://docs.djangoproject.com/en/4.1/ref/paginator/.
- Введение в представления на основе классов: https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/.
- Отправка электронных писем с помощью Django: https://docs.djangoproject.com/en/4.1/topics/email/.
- Типы полей формы в Django: https://docs.djangoproject.com/en/4.1/ref/forms/fields/.
- Работа с формами: https://docs.djangoproject.com/en/4.1/topics/forms/.
- Создание форм из моделей: https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/.
- Модельные взаимосвязи многие-к-одному: https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/.

Резюме

В данной главе вы научились определять для моделей канонические URL-адреса. Вы создали дружественные для поисковой оптимизации URL-адреса постов блога и реализовали постраничную разбивку объектов для списка постов. Вы также научились работать с формами Django и моделировать формы. Вы создали систему рекомендации постов по электронной почте и систему комментариев для своего блога.

В следующей главе вы создадите в своем блоге систему тегирования. Вы научитесь формировать сложные наборы запросов, чтобы извлекать объекты по сходству. Вы освоите создание конкретно-прикладных шаблонных тегов и фильтров. Вы также разработаете конкретно-прикладную карту сайта и новостную ленту для постов блога и реализуете функциональность полнотекстового поиска постов.