Geekbrains

**Разработка сайта для ведения профессионального блога с использованием фреймворка Django**

Программа:

Цифровые Профессии

Специализация:

Python-разработчик

ФИО

Морозов Владислав Вячеславович

Саратов

2024

Оглавление

[Введение 4](#_Toc166505256)

[Теоретическая и практическая главы 7](#_Toc166505257)

[Теория 7](#_Toc166505258)

[Практика 7](#_Toc166505259)

[Создание виртуальной среды и установка Dango 8](#_Toc166505260)

[Создание проекта 8](#_Toc166505261)

[blog 8](#_Toc166505262)

[mysite 10](#_Toc166505263)

[Создание моделей данных блога 11](#_Toc166505264)

[class PublishedManager 11](#_Toc166505265)

[Модель Post 12](#_Toc166505266)

[Модель Comment 15](#_Toc166505267)

[Активация приложения 16](#_Toc166505268)

[Создание и применение миграций 16](#_Toc166505269)

[Создание сайта администрирования для моделей 17](#_Toc166505270)

[Создание суперпользователя 17](#_Toc166505271)

[Добавление моделей на сайт администрирования 17](#_Toc166505272)

[Адаптация внешнего вида моделей под конкретно-прикладную задачу 17](#_Toc166505273)

[Разработка представлений 19](#_Toc166505274)

[Функция вызова опубликованных постов 20](#_Toc166505275)

[Функция вызова информации об одном посте 21](#_Toc166505276)

[Функция отправки пост по его идентификатору id 23](#_Toc166505277)

[Функция 25](#_Toc166505278)

[Добавление шаблонов URL-адресов представлений 27](#_Toc166505279)

[\mysite\blog\urls.py 27](#_Toc166505280)

[\mysite\mysite\urls.py 28](#_Toc166505281)

[Создание шаблонов представлений 29](#_Toc166505282)

[pagination.html 30](#_Toc166505283)

[base.html 31](#_Toc166505284)

[comment.html 33](#_Toc166505285)

[comment\_form.html 33](#_Toc166505286)

[detail.html 34](#_Toc166505287)

[latest\_posts.html 36](#_Toc166505288)

[list.html 37](#_Toc166505289)

[pagination.html 38](#_Toc166505290)

[share.html 39](#_Toc166505291)

[comment\_form.html 40](#_Toc166505292)

[Заключение 43](#_Toc166505293)

[Список используемой литературы 44](#_Toc166505294)

[Пример 44](#_Toc166505295)

[Приложения 45](#_Toc166505296)

# Введение

Профессиональные блоги играют значительную роль в современном мире бизнеса и маркетинга. Они представляют собой платформу для экспертов, компаний и организаций, чтобы делиться информацией, идеями, исследованиями, опытом и мнениями в отрасли. Ниже представлен краткий обзор некоторых ключевых аспектов профессиональных блогов:

1. **Экспертное мнение**: Профессиональные блоги предоставляют площадку для экспертов в различных областях (технологии, маркетинг, финансы, здравоохранение и т. д.) для выражения своих мнений, деления своим профессиональным опытом и знаниями.

2. **Информационный ресурс**: Блоги могут служить важным источником информации для профессионалов, предоставляя им последние новости, обзоры индустрии, аналитику и советы от экспертов.

3. **Контент-маркетинг**: для компаний блоги часто являются частью стратегии контент-маркетинга. Целенаправленное создание интересного, информативного и ценного контента помогает привлекать аудиторию, укреплять бренд и устанавливать авторитет на рынке.

4. **Обратная связь и взаимодействие**: Блоги предоставляют возможность для комментариев, обсуждений и обратной связи со стороны читателей, что способствует формированию сообщества и укреплению отношений с аудиторией.

5. **Личный бренд**: для профессионалов блоги представляют собой возможность развития личного бренда. Публикация полезного контента, выражение экспертного мнения и участие в диалогах помогают устанавливать авторитет и расширять профессиональные связи.

6. **SEO и привлечение трафика**: Качественный и оптимизированный контент на блоге способствует улучшению показателей поисковой выдачи, привлечению целевой аудитории и увеличению трафика на сайт.

7. **Обучающий ресурс**: В блогах можно размещать обучающие материалы, уроки, кейс-стади и другой образовательный контент, который приносит ценность и помогает читателям осваивать новые знания и навыки.

8. **Монетизация**: Некоторые профессиональные блоги становятся источником дохода для их владельцев через размещение платного контента, рекламу, партнерские программы и другие стратегии.

9. **Брендирование**: Блоги часто используются для укрепления бренда компании или персонального бренда, позволяя демонстрировать ценности, миссию компании, а также создавать узнаваемый образ и стиль.

10. **Исследования и тенденции**: Регулярные публикации на блоге могут отражать изменения в отрасли, тенденции развития рынка и новые направления, демонстрируя способность автора следить за индустрией.

Профессиональные блоги играют важную роль в обмене знаниями, создании ценного контента и укреплении профессионального сообщества, предоставляя значительные возможности для бизнеса, маркетинга, личного бренда и развития отрасли.

Разработка блога на платформе Django представляет собой увлекательный процесс, включающий создание моделей для представления данных (таких как статьи, комментарии и пользователи), настройку маршрутов URL для страниц блога, разработку шаблонов для отображения контента и использование ORM (Object-Relational Mapping) для взаимодействия с базой данных.

Например, для создания модели статьи в блоге, можно определить класс Article с полями, представляющими заголовок, текст, дату публикации и автора. Затем, с использованием встроенной административной панели Django, разработчик может добавлять, редактировать и удалять статьи.

Для обработки запросов и отображения страниц блога, Django использует представления (views), которые могут быть написаны на Python и возвращать HTTP-ответы с помощью шаблонов HTML.

Также, Django предоставляет удобные инструменты для аутентификации пользователей, управления комментариями, обработки изображений и обеспечения безопасности приложения.

В целом, разработка блога на Django позволяет создать мощное и масштабируемое веб-приложение с удобным администрированием, гибкой настройкой пользовательского интерфейса и высокой производительностью.

# Теоретическая и практическая главы

## Теория

Разработка блога на платформе Django представляет собой увлекательный процесс, включающий создание моделей для представления данных (таких как статьи, комментарии и пользователи), настройку маршрутов URL для страниц блога, разработку шаблонов для отображения контента и использование ORM (Object-Relational Mapping) для взаимодействия с базой данных.

Например, для создания модели статьи в блоге, можно определить класс Article с полями, представляющими заголовок, текст, дату публикации и автора. Затем, с использованием встроенной административной панели Django, разработчик может добавлять, редактировать и удалять статьи.

Для обработки запросов и отображения страниц блога, Django использует представления (views), которые могут быть написаны на Python и возвращать HTTP-ответы с помощью шаблонов HTML.

Также, Django предоставляет удобные инструменты для аутентификации пользователей, управления комментариями, обработки изображений и обеспечения безопасности приложения.

В целом, разработка блога на Django позволяет создать мощное и масштабируемое веб-приложение с удобным администрированием, гибкой настройкой пользовательского интерфейса и высокой производительностью.

## Практика

### Создание виртуальной среды и установка Dango

### Создание проекта

#### blog

Структура папок директории blog

| admin.py

| apps.py

| blog.doc

| feeds.py

| forms.py

| models.py

| sitemaps.py

| tests.py

| urls.py

| views.py

| \_\_init\_\_.py

|

+---migrations

| | 0001\_initial.py

| | 0002\_alter\_post\_slug.py

| | 0003\_comment.py

| | 0004\_post\_tags.py

| | \_\_init\_\_.py

| |

| \---\_\_pycache\_\_

| 0001\_initial.cpython-312.pyc

| 0002\_alter\_post\_slug.cpython-312.pyc

| 0003\_comment.cpython-312.pyc

| 0004\_post\_tags.cpython-312.pyc

| \_\_init\_\_.cpython-312.pyc

|

+---static

| \---css

| blog.css

|

+---templates

| | pagination.html

| |

| \---blog

| | base.html

| |

| \---post

| | comment.html

| | comment\_form.html

| | detail.html

| | latest\_posts.html

| | list.html

| | pagination.html

| | share.html

| |

| \---includes

| comment\_form.html

|

+---templatetags

| | blog\_tags.py

| | \_\_init\_\_.py

| |

| \---\_\_pycache\_\_

| blog\_tags.cpython-312.pyc

| \_\_init\_\_.cpython-312.pyc

|

\---\_\_pycache\_\_

admin.cpython-312.pyc

apps.cpython-312.pyc

feeds.cpython-312.pyc

forms.cpython-312.pyc

models.cpython-312.pyc

sitemaps.cpython-312.pyc

urls.cpython-312.pyc

views.cpython-312.pyc

\_\_init\_\_.cpython-312.pyc

#### mysite

Структура папок тома директории mysite

| asgi.py

| mysite.doc

| settings.py

| urls.py

| wsgi.py

| \_\_init\_\_.py

|

\---\_\_pycache\_\_

settings.cpython-312.pyc

urls.cpython-312.pyc

wsgi.cpython-312.pyc

\_\_init\_\_.cpython-312.pyc

#### \mysite\settings.py

from pathlib import Path

# Build paths inside the project like this: BASE\_DIR / 'subdir'.

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/5.0/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = 'django-insecure-)nvxq#-5tez(@erq1^zplwhu3h4)6+@la1oot4^q\*rt\_prul&e'

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog.apps.BlogConfig',

'taggit',

'django.contrib.sites',

'django.contrib.sitemaps',

]

SITE\_ID = 1

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'mysite.urls'

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

WSGI\_APPLICATION = 'mysite.wsgi.application'

# Database

# https://docs.djangoproject.com/en/5.0/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

# Password validation

# https://docs.djangoproject.com/en/5.0/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

{

'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

},

]

# Internationalization

# https://docs.djangoproject.com/en/5.0/topics/i18n/

LANGUAGE\_CODE = 'ru-ru'

TIME\_ZONE = 'UTC'

USE\_I18N = True

USE\_TZ = True

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/5.0/howto/static-files/

STATIC\_URL = 'static/'

# Default primary key field type

# https://docs.djangoproject.com/en/5.0/ref/settings/#default-auto-field

# DEFAULT\_AUTO\_FIELD = 'django.db.models.BigAutoField'

# Настройка почты

# EMAIL\_BACKEND = 'django.core.mail.backends.console.EmailBackend'

EMAIL\_HOST = 'smtp.yandex.ru'

EMAIL\_PORT = 587

EMAIL\_HOST\_USER = "morozovvladislav@yandex.ru"

EMAIL\_HOST\_PASSWORD = "ycigmjyqzvxycytz"

EMAIL\_USE\_TLS = True

EMAIL\_USE\_SSL = False

SERVER\_EMAIL = EMAIL\_HOST\_USER

DEFAULT\_FROM\_EMAIL = EMAIL\_HOST\_USER

### Создание моделей данных блога

Описание моделей находится в файле \mysite\blog\models.py

from django.db import models

from django.utils import timezone

from django.contrib.auth.models import User

from django.urls import reverse

from taggit.managers import TaggableManager

* **Djngo.Utils.Timezone** является вспомогательной системой в Django, которая хранит данные и информацию о времени в UTC в базе данных.
* Встроенный в Django фреймворк аутентификации располагается в пакете **django.contrib.auth** и содержит модель User (Пользователь). Модель User будет применяться из указанного фреймворка аутентификации, чтобы создавать взаимосвязи между пользователями и постами
* **reverse**: Резольвер URL-адресов – это программная утилита или функция, которая конвертирует логический адрес или метаданные в физический URL-адрес целевых данных

#### class PublishedManager

class PublishedManager(models.Manager):

def get\_queryset(self):

return super().get\_queryset().filter(status=Post.Status.PUBLISHED)

Конкретно-прикладной менеджер, чтобы извлекать все посты, имеющие статус PUBLISHED

#### Модель Post

class Post(models.Model):

class Status(models.TextChoices):

DRAFT = 'DF', 'Draft'

PUBLISHED = 'PB', 'Published'

title = models.CharField(max\_length=250)

slug = models.SlugField(max\_length=250, unique\_for\_date='publish')

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)

tags = TaggableManager()

status = models.CharField(

max\_length=2, choices=Status.choices, default=Status.DRAFT)

objects = models.Manager() # менеджер, применяемый по умолчанию

published = PublishedManager() # конкретно-прикладной менеджер

class Meta:

ordering = ['-publish']

indexes = [

models.Index(fields=['-publish']),

]

def \_\_str\_\_(self):

return self.title

* **Перечисляемый класс Status** определен путем подклассирования класса models.TextChoices. Доступными вариантами статуса поста являются DRAFT и PUBLISHED. Их соответствующими значениями выступают DF и PB, а их метками или читаемыми именами являются Draft и Published.
* **title**: поле заголовка поста. Это поле с типом CharField, которое транслируется в столбец VARCHAR в базе данных SQL
* **slug**: поле SlugField, которое транслируется в столбец VARCHAR в базе данных SQL. Слаг – это короткая метка, содержащая только буквы, цифры, знаки подчеркивания или дефисы
* **author**: создана взаимосвязь между пользователями и постами, которая будет указывать на конкретных пользователей и написанные ими посты
* **body**: поле для хранения тела поста. Это поле с типом TextField, которое транслируется в столбец Text в базе данных SQL
* **publish**: поле с типом DateTimeField, которое транслируется в столбец DATETIME в базе данных SQL. Оно будет использоваться для хранения даты и времени публикации поста. По умолчанию значения поля за даются методом Django timezone.now.
* **created:** поле с типом DateTimeField. Оно будет использоваться для хранения даты и времени создания поста. При применении параметра auto\_now\_add дата будет сохраняться автоматически во время создания объекта.
* **updated**: поле с типом DateTimeField. Оно будет использоваться для хранения последней даты и времени обновления поста. При применении параметра auto\_now дата будет обновляться автоматически во время сохранения объекта
* **tags**: позволит добавлять, извлекать и удалять теги из объектов Post
* **status:** экземпляр типа CharField. Оно содержит параметр choices, чтобы ограничивать значение поля вариантами из Status.choices. Кроме того, применяя параметр default, задано значение поля, которое будет использоваться по умолчанию. В этом поле статус DRAFT используется в качестве предустановленного варианта, если не указан иной.
* **objects**: стандартный менеджер
* **published:** конкретно-прикладной менеджер
* **class Meta:** Этот класс определяет метаданные модели. Мы используем атрибут ordering, сообщающий Django, что он должен сортировать результаты по полю publish
  + **ordering**, сообщающий Django, что он должен сортировать результаты по полю publish
  + **indexes:** опция позволяет определять в модели индексы базы данных, которые могут содержать одно или несколько полей в возрастающем либо убывающем порядке, или функциональные выражения и функции базы данных
* **\_\_str\_\_:** метод Python, который применяется по умолчанию и возвращает строковый литерал с удобочитаемым представлением объекта
* **get\_absolute\_url:**

def get\_absolute\_url(self):

return reverse('blog:post\_detail', args=[self.publish.year,

self.publish.month,

self.publish.day,

self.slug]

)

Функция reverse() будет формировать URL-адрес динамически, применяя имя URL-адреса, определенное в  шаблонах URL-адресов.

#### Модель Comment

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}'

* **post:** Поле ForeignKey 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.
* **active**: управляет 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.
* **created:** хранит дату и время создания комментария. Используя auto\_now\_add, дата будет сохраняться автоматически при создании объекта.
* **class Meta:** 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.

### Активация приложения

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog.apps.BlogConfig',

'taggit',

'django.contrib.sites',

'django.contrib.sitemaps',

]

### Создание и применение миграций

Для моделей блога, необходимо создать соответствующие таблицу базы данных. Django идет в комплекте с системой миграции, которая отслеживает внесенные в модели изменения и позволяет их распространять по базе данных.

python manage.py makemigrations blog

python manage.py migrate

### Создание сайта администрирования для моделей

#### Создание суперпользователя

python manage.py createsuperuser

#### Добавление моделей на сайт администрирования

#### Адаптация внешнего вида моделей под конкретно-прикладную задачу

Содержится в файле \mysite\blog\admin.py

from django.contrib import admin

from .models import Post, Comment

* **Django admin** — это приложение, которое может использовать модели для автоматического создания части сайта, предназначенной для создания, просмотра, обновления и удаления записей.

**class PostAdmin**

@admin.register(Post)

class PostAdmin(admin.ModelAdmin):

list\_display = ['title', 'slug', 'author', 'publish', 'status']

list\_filter = ['status', 'created', 'publish', 'author']

search\_fields = ['title', 'body']

prepopulated\_fields = {'slug': ('title',)}

raw\_id\_fields = ['author']

date\_hierarchy = 'publish'

ordering = ['status', 'publish']

* **list\_display** позволяет задавать поля модели, которые вы хотите показывать на странице списка объектов администрирования
* **list\_filter** поля для фильтрования результатов
* **search\_fields** список полей, по которым можно вести поиск
* **prepopulated\_fields** функция, которая позволяет автоматически заполнять одно или несколько полей на основе значения другого поля
* **raw\_id\_fields**
* **date\_hierarchy:** навигационные ссылки для навигации по иерархии дат

**class CommentAdmin**

@admin.register(Comment)

class CommentAdmin(admin.ModelAdmin):

list\_display = ['name', 'email', 'post', 'created', 'active']

list\_filter = ['active', 'created', 'updated']

search\_fields = ['name', 'email', 'body']

### Разработка представлений

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

from taggit.models import Tag

from django.db.models import Count

* **render:** функция сокращенного доступа принимает объект request, путь к шаблону и контекстные переменные, чтобы прорисовать данный шаблон. Она возвращает объект HttpResponse с прорисованным текстом (обычно исходным кодом HTML). Функция сокращенного доступа render() учитывает контекст запроса, поэтому любая переменная, установленная процессорами контекста шаблона, доступна данному шаблону.
* **get\_object\_or\_404:** функция сокращенного доступа извлекает объект, соответствующий переданным параметрам, либо исключение HTTP с кодом состояния, равным 404 (не найдено), если объект не найден
* **redirect:**
* **Paginator:**
* **EmptyPage:**
* **PageNotAnInteger:**
* **ListView:**
* **EmailPostForm:**
* **CommentForm:**
* **send\_mail:**
* **require\_POST:**
* **Tag:**
* **Count:**

#### Функция вызова опубликованных постов

def post\_list(request, tag\_slug=None):

'''

Функция вызова опубликованных постов

'''

post\_list = Post.published.all()

tag = None

if tag\_slug:

tag = get\_object\_or\_404(Tag, slug=tag\_slug)

post\_list = post\_list.filter(tags\_\_in=[tag])

# 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 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

posts = paginator.page(paginator.num\_pages)

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

Представление post\_list принимает объект request в качестве единственного параметра. Указанный параметр необходим для всех функций-представлений. В данном представлении извлекаются все посты со статусом PUBLISHED, используя менеджер published

#### Функция вызова информации об одном посте

def post\_detail(request, year, month, day, post):

'''

Информация об одном посте

'''

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()

# Список схожих постов

post\_tags\_ids = post.tags.values\_list('id', flat=True)

similar\_posts = Post.published.filter(

tags\_\_in=post\_tags\_ids).exclude(id=post.id)

similar\_posts = similar\_posts.annotate(same\_tags=Count(

'tags')).order\_by('-same\_tags', '-publish')[:4]

return render(request, 'blog/post/detail.html', {'post': post, 'comments': comments, 'form': form, 'similar\_posts': similar\_posts})

Представление детальной информации о посте. Указанное, представление принимает аргумент id поста

* набор запросов QuerySet, чтобы извлекать все активные комментарии к посту, как показано ниже:

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

* Используется объект post, чтобы извлекать связанные объекты Comment. Применен менеджер comments для ранее определенных в модели Comment связанных с Comment объектов, используя атрибут related\_name поля ForeignKey в модели Post;
* Cсоздан экземпляр формы для комментария посредством инструкции:

form = CommentForm()

#### Функция отправки пост по его идентификатору id

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, 'morozovvladislav@yandex.ru',

[cd['to']])

sent = True

else:

form = EmailPostForm()

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

'form': form, 'sent': sent})

Представление post\_share в качестве параметров принимает объект request и переменную post\_id. Функцию сокращенного доступа get\_object\_or\_404() извлекает 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.

В приведенном выше исходном коде объявлена переменная sent с начальным значением False. Эта переменная получает значение True после отправки электронного письма. Переменную sent используется в шаблоне отображения сообщения об успехе при успешной передаче формы. Ссылка на пост должна вставляться в электронное письмо, путем получения абсолютного пути к посту методом get\_absolute\_url(). Этот путь используется на входе в метод request.build\_absolute\_url(), чтобы сформировать полный URL-адрес, включая HTTP-схему и хост-имя (hostname). Тема и текст сообщения электронного письма создается, используя очищенные данные валидированной формы. Электронное письмо отправляется на адрес электронной почты, указанный в поле to (Кому) формы.

#### Функция

@ 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})

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 POSTданные, 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.

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.

### Добавление шаблонов URL-адресов представлений

#### \mysite\blog\urls.py

from django.urls import path

from . import views

from .feeds import LatestPostsFeed

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'),

path('<int:post\_id>/share/', views.post\_share, name='post\_share'),

path('<int:post\_id>/comment/', views.post\_comment, name='post\_comment'),

path('tag/<slug:tag\_slug>/', views.post\_list, name='post\_list\_by\_tag'),

path('feed/', LatestPostsFeed(), name='post\_feed'),

]

Шаблоны URL-адресов позволяют соотносить URL-адреса с представлениями. Шаблон URL-адреса состоит из строкового шаблона, представления и, опционально, имени, которое позволяет именовать URL-адрес в масштабе всего проекта. Django просматривает каждый шаблон URL-адреса и останавливается на первом, который совпадает с запрошенным URL-адресом. Затем Django импортирует представление, совпадающее с шаблоном URL-адреса, и исполняет его, передавая экземпляр класса HttpRequest и именованные или позиционные аргументы.

Именное пространство приложения определяется с помощью переменной app\_name.

#### \mysite\mysite\urls.py

from django.contrib import admin

from django.urls import path, include

from django.contrib.sitemaps.views import sitemap

from blog.sitemaps import PostSitemap

sitemaps = {

'posts': PostSitemap,

}

urlpatterns = [

path('admin/', admin.site.urls),

path('blog/', include('blog.urls', namespace='blog')),

path('sitemap.xml', sitemap, {'sitemaps': sitemaps},

name='django.contrib.sitemaps.views.sitemap')

]

### Создание шаблонов представлений

Структура папок директории templates

| pagination.html

| tampates.doc

|

\---blog

| base.html

|

\---post

| comment.html

| comment\_form.html

| detail.html

| latest\_posts.html

| list.html

| pagination.html

| share.html

|

\---includes

comment\_form.html

Django обладает мощным языком шаблонов, который позволяет указывать внешний вид отображения данных. Он основан на шаблонных тегах, шаблонных переменных и шаблонных фильтрах:

• шаблонные теги управляют прорисовкой шаблона и выглядят как {% tag %};

• шаблонные переменные заменяются значениями при прорисовке шаблона и выглядят как {{ variable }};

• шаблонные фильтры позволяют видоизменять отображаемые переменные и выглядят как {{ variable|filter }}.

#### pagination.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 предыдущую и следующую ссылки, а также отображать текущую страницу и общее число страниц результатов

#### base.html

{% load blog\_tags %}

{% load static %}

<!DOCTYPE html>

<html>

<head>

<title>{% block title %}{% endblock %}</title>

<link href="{% static 'css/blog.css' %}" rel="stylesheet">

</head>

<body>

<div id="content">

{% block content %}

{% endblock %}

</div>

<div id="sidebar">

<h2>My blog</h2>

<p>

This is my blog.

I've written {% total\_posts %} posts so far.

</p>

<p>

<a href="{% url 'blog:post\_feed' %}">

Subscribe to my RSS feed

</a>

</p>

<h3>Latest posts</h3>

{% show\_latest\_posts 3 %}

<h3>Most commented posts</h3>

{% get\_most\_commented\_posts as most\_commented\_posts %}

<ul>

{% for post in most\_commented\_posts %}

<li>

<a href="{{ post.get\_absolute\_url }}">{{ post.title }}</a>

</li>

{% endfor %}

</ul>

</div>

</body>

</html>

Тег {% load static %} сообщает Django, что нужно загрузить статические шаблонные теги (static), предоставляемые приложением django.contrib.staticfiles, которое содержится в настроечном параметре INSTALLED\_APPS. После их загрузки шаблонный тег {% static %} можно использовать во всем этом шаблоне. С помощью указанного шаблонного тега можно вставлять статические файлы, такие как файл blog.css.

Два тега {% block %}. Они сообщают Django, что нужно определить блок в отмеченной области. Определен блок под названием title и блок под названием content.

#### comment.html

{% 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 %}

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 %}.

#### comment\_form.html

<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>

Динамически формируется URL-адрес action HTML-элемента <form>, используя шаблонный тег {% url %}. Формируется URL-адрес представления post\_comment, которое будет обрабатывать форму.

Тег {% csrf\_token %} необходим, чтобы защититься от CSRF, поскольку данная форма будет

передаваться на обработку методом POST.

#### detail.html

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

{% load blog\_tags %}

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

{% block content %}

<h1>{{ post.title }}</h1>

<p class="date">

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

</p>

{{ post.body|markdown }}

<p>

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

Share this post

</a>

</p>

<h2>Similar posts</h2>

{% for post in similar\_posts %}

<p>

<a href="{{ post.get\_absolute\_url }}">{{ post.title }}</a>

</p>

{% empty %}

There are no similar posts yet.

{% endfor %}

{% with comments.count as total\_comments %}

<h2>

{{ total\_comments }} comment{{ total\_comments|pluralize }}

</h2>

{% endwith %}

{% for comment in comments %}

<div class="comment">

<p class="info">

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

{{ comment.created }}

</p>

{{ comment.body|linebreaks }}

</div>

{% empty %}

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

{% endfor %}

{% include "blog/post/includes/comment\_form.html" %}

{% endblock %}

Шаблонный тег {% extends %} сообщает Django, что надо наследовать от шаблона blog/base.html. Затем заполняются блоки title и content базового шаблона.

* Используем Django ORM-преобразователь, применяя набор запросов comments.count(). Тег {% with %} позволяет присваивать значение новой переменной, которая будет доступна шаблоне до тех пор, пока не появится тег {% endwith %}.
* Используем шаблонный фильтр pluralize, чтобы отображать суффикс множественного числа для слова comment, в зависимости от значения total\_comments.
* 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.

#### latest\_posts.html

<ul>

{% for post in latest\_posts %}

<li>

<a href="{{ post.get\_absolute\_url }}">{{ post.title }}</a>

</li>

{% endfor %}

</ul>

В приведенном выше исходном коде отображается неупорядоченный список постов, используя возвращаемую вашим шаблонным тегом переменную latest\_posts.

#### list.html

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

{% load blog\_tags %}

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

{% block content %}

<h1>My Blog</h1>

{% if tag %}

<h2>Posts tagged with "{{ tag.name }}"</h2>

{% endif %}

{% for post in posts %}

<h2>

<a href="{{ post.get\_absolute\_url }}">

{{ post.title }}

</a>

</h2>

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

<p class="date">

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

</p>

{{ post.body|markdown|truncatewords\_html:30 }}

{% endfor %}

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

{% endblock %}

Шаблонный тег {% extends %} сообщает Django, что надо наследовать от шаблона blog/base.html. Затем заполняются блоки title и content базового шаблона. Посты прокручиваются в цикле, и их заголовок, дата, автор и тело отображаются на странице, включая ссылку в заголовке на подробный URL-адрес поста. веб-фреймворком Django шаблонного тега {% url %}.

Шаблонный тег {% url %} позволяет формировать URL-адреса динамически по их имени. Мы используем blog:post\_detail, чтобы ссылаться на URL-адрес post\_detail в именном пространстве blog. Мы передаем необходимый параметр post.id, чтобы сформировать URL-адрес для каждого поста.

В теле поста применяются два шаблонных фильтра: truncatewords усекает значение до указанного числа слов, а linebreaks конвертирует результат в разрывы строк в формате HTML. При этом можно конкатенировать столько шаблонных фильтров, сколько потребуется; каждый из них будет применен к результату, сгенерированному предыдущим.

Шаблонный тег {% include %} загружает шаблон pagination.html 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.

#### pagination.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>

#### share.html

{% 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 %}

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 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 %}

Добавлен шаблонный тег {% csrf\_token %}. Указанный тег вводит скрытое поле с автоматически сгенерированным токеном во избежание атак по подделке межсайтовых запросов (CSRF).

#### comment\_form.html

<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>

### Разработка форм с помощью Django

from .models import Comment

from django import forms

#### Форма для отсылки поста по почте

class EmailPostForm(forms.Form):

name = forms.CharField(max\_length=25)

email = forms.EmailField()

to = forms.EmailField()

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

• **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.

#### Форма для комментариев

class CommentForm(forms.ModelForm):

class Meta:

model = Comment

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

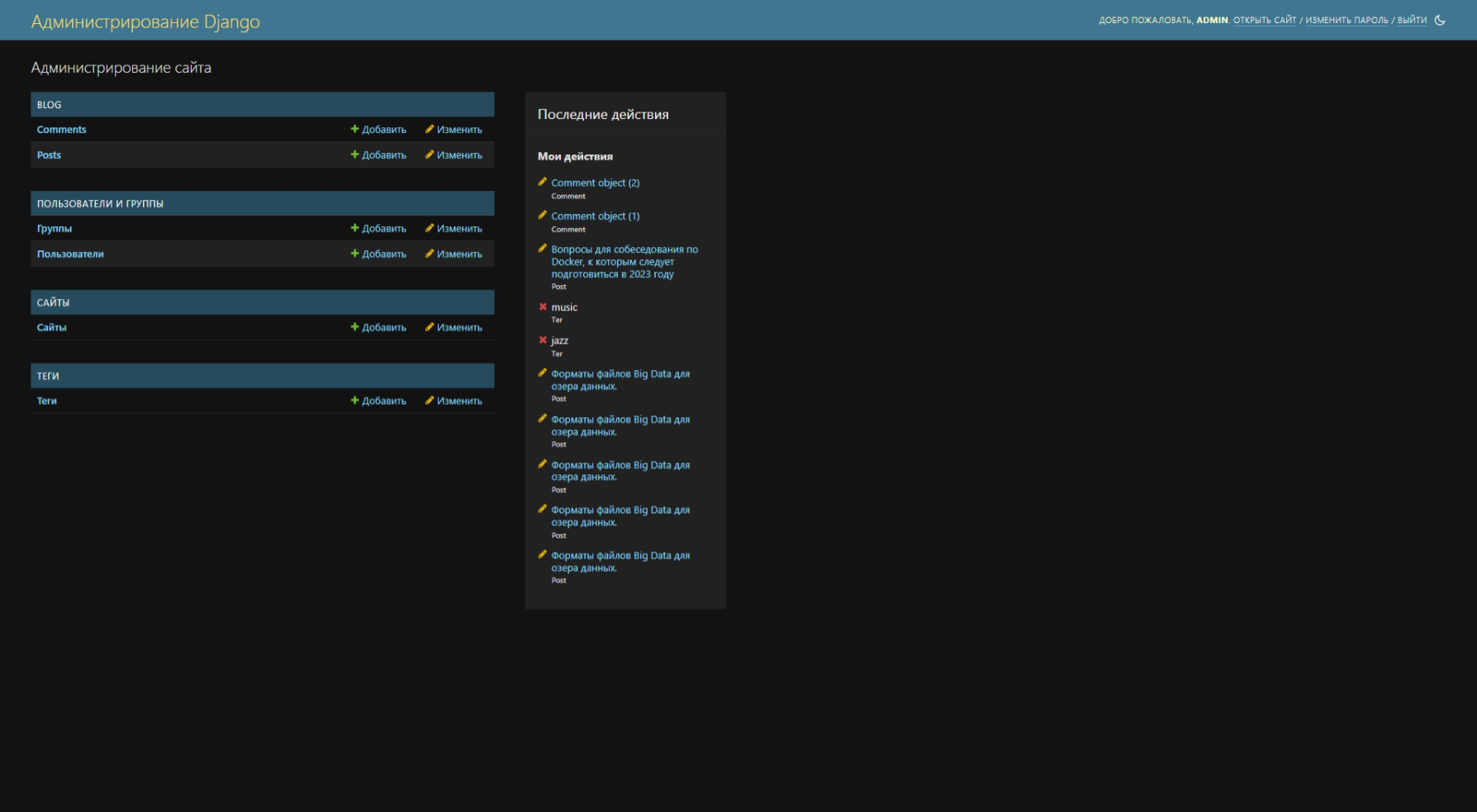
# Заключение

Использование фреймворка Django позволило за короткий срок разработать качественный продукт. Блог позволяет отображать список постов и детальную информацию по каждому. Каждый пост снабжён тегами, благодаря чему можно подобрать посты наиболее близкие по содержанию. Понравившийся пост можно рекомендовать другу путем направления электронного сообщения. Посты имеют дружественные для поисковой оптимизации URL-адреса. Для удобства использования предусмотрена постраничная разбивка списка постов. Новости транслируются в RSS. На каждый пост пользователь может писать комментарии, при этом предусмотрено отображение наиболее обсуждаемых постов.

# Список используемой литературы

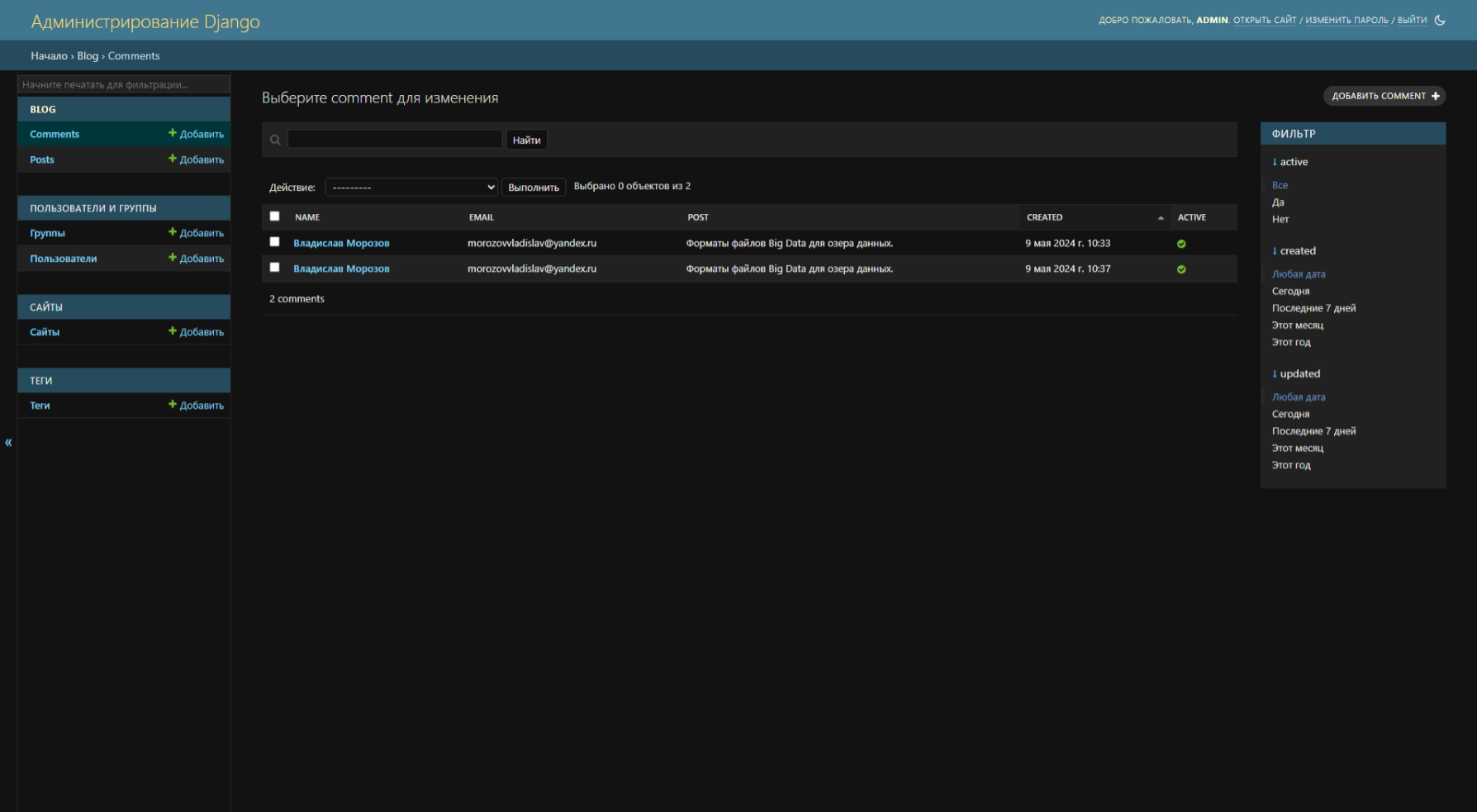
1. Хекслет – Почему Django — лучший фреймворк для разработки сайтов [Электронный <https://ru.hexlet.io/blog/posts/pochemu-django-luchshiy-freymvork-dlya-razrabotki-saytov> , свободный
2. WebforMyself – Django — лучший фреймворк для разработки сайтов [Электронныйресурс] – Режим доступа: <https://webformyself.com/django-luchshij-frejmvork-dlya-razrabotki-sajtov/> , свободный
3. ArtKiev – Что такое Django Framework и для чего он применяется [Электронный ресурс] – Режим доступа: <https://artkiev.com/blog/django-framework.htm> , свободный (дата обращения: 25.03.2021)
4. Django – Django Documentation [Электронный ресурс] – Режим доступа: <https://docs.djangoproject.com/en/3.1/>,

# Приложения

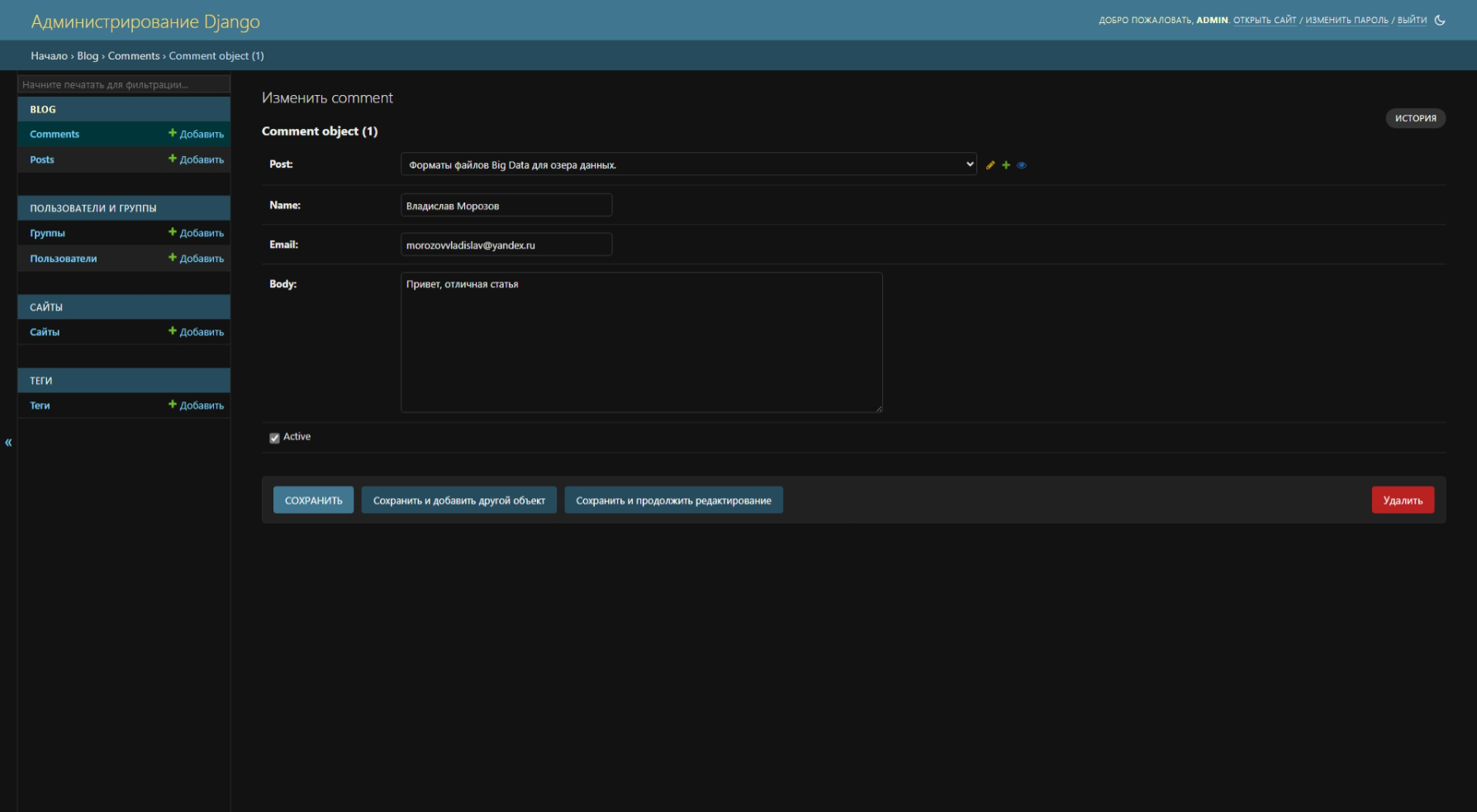


## Администрирование Сайта

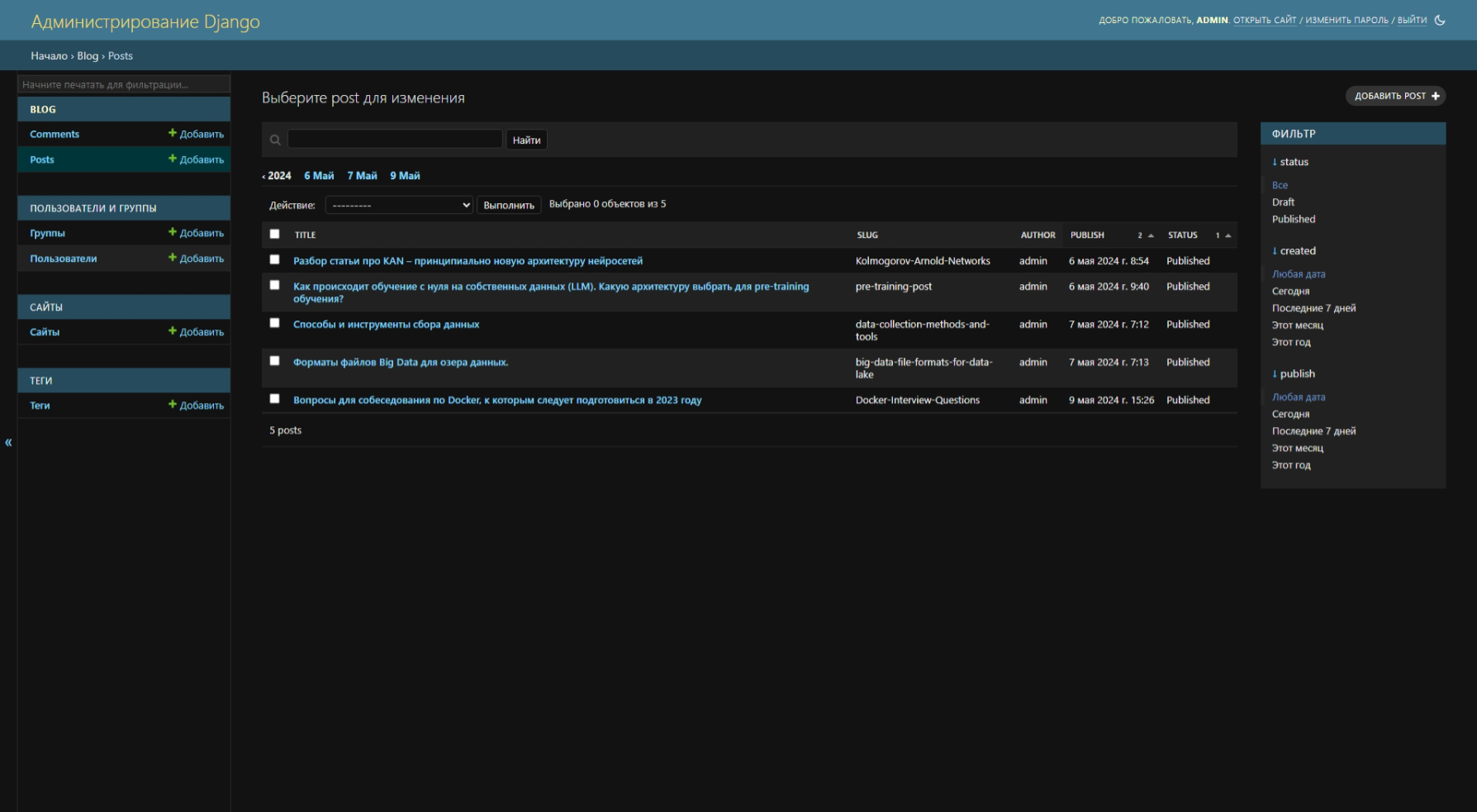
### Выбор модели



### Список комментариев

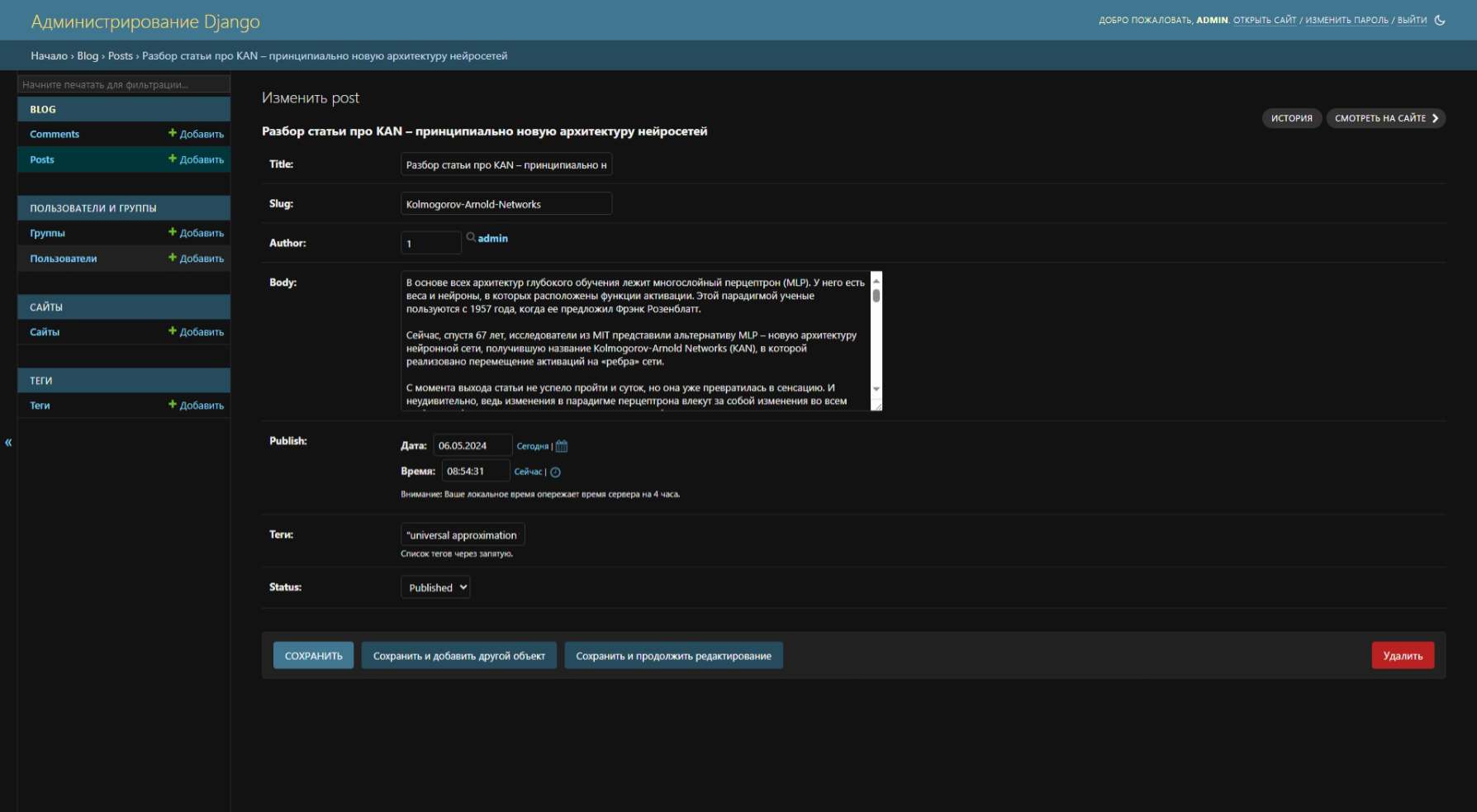


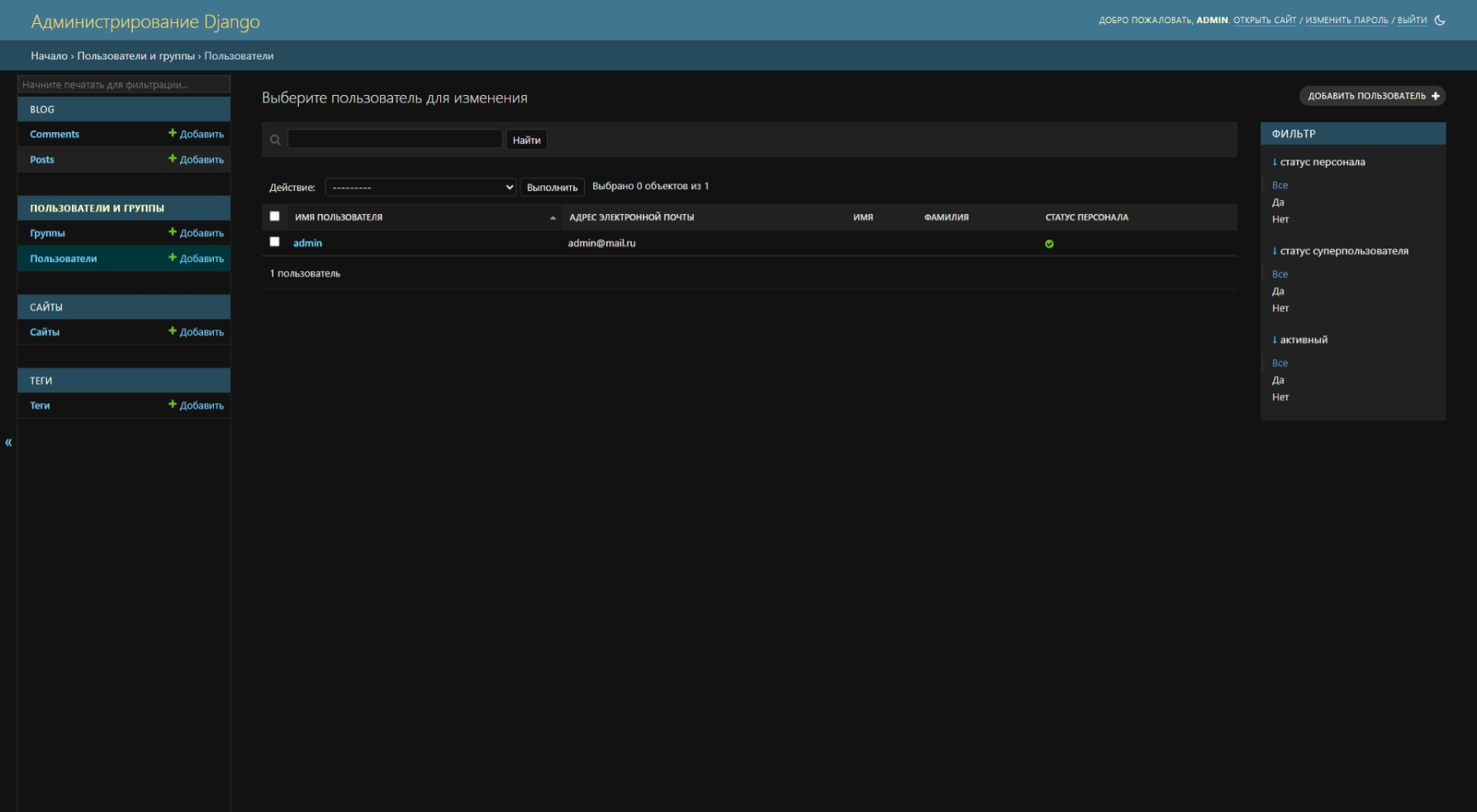
### Редактирование комментариев



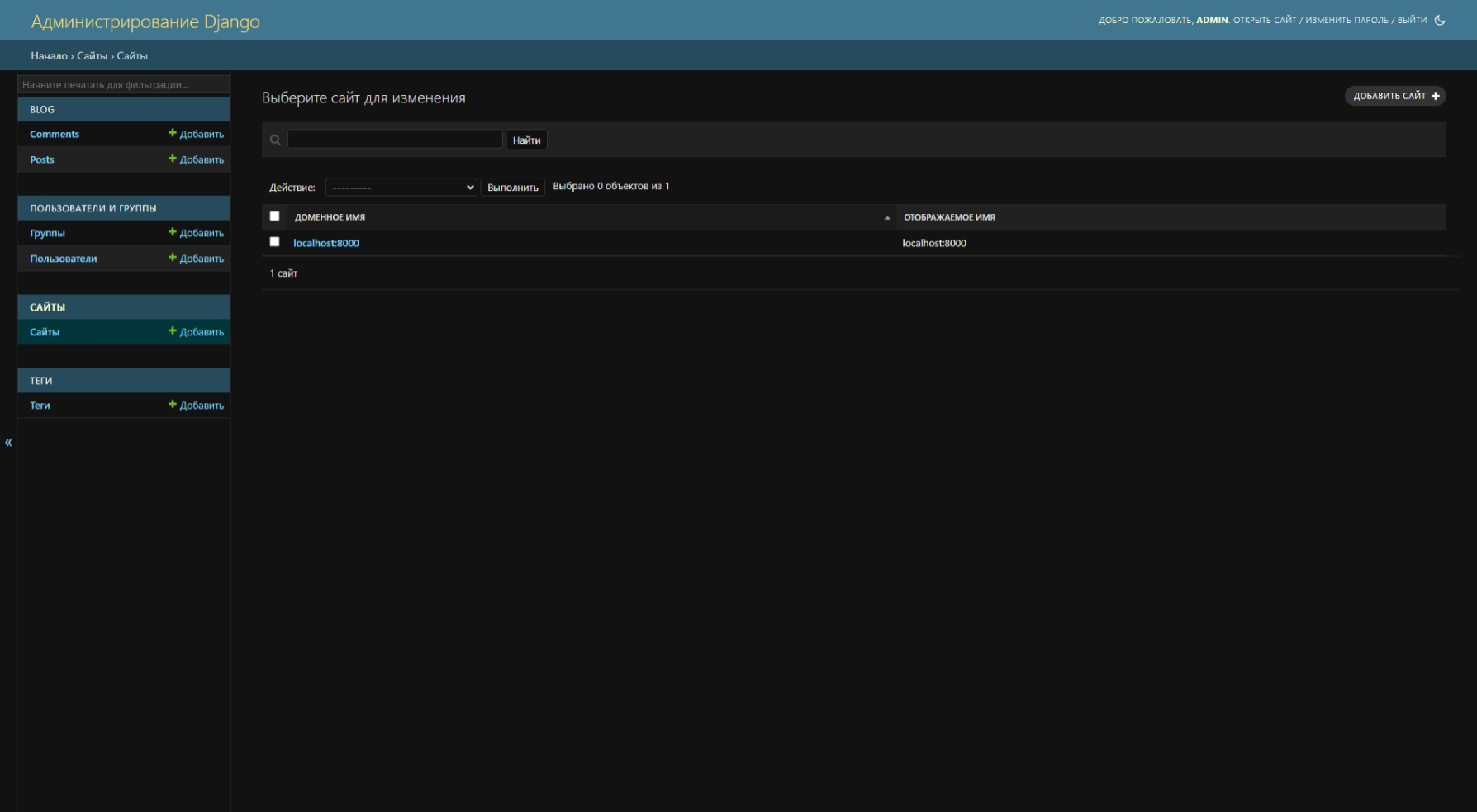
### Список постов

### Редактирование поста



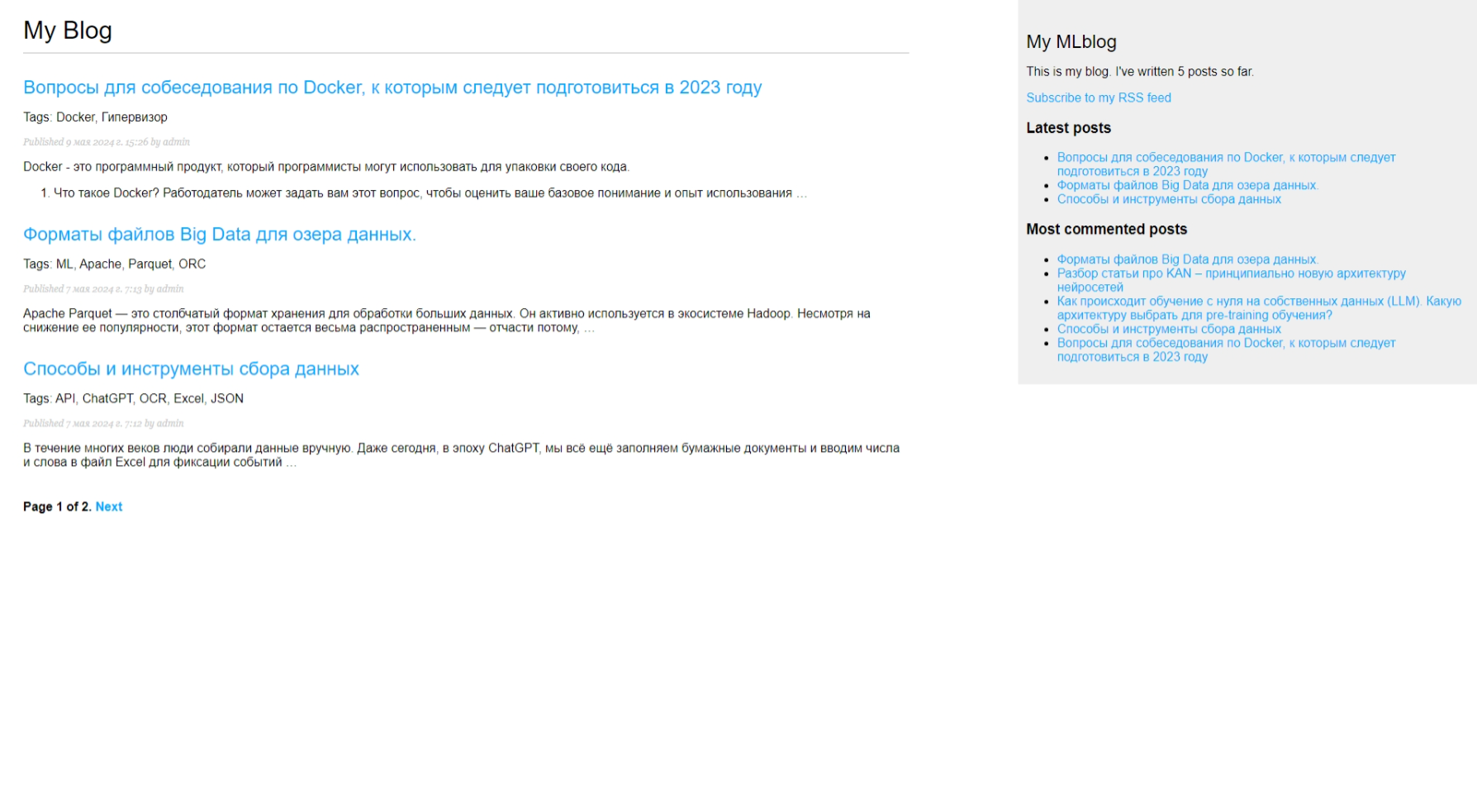


### Редактирование информации о пользователе



### Выбор сайта

### Список тегов

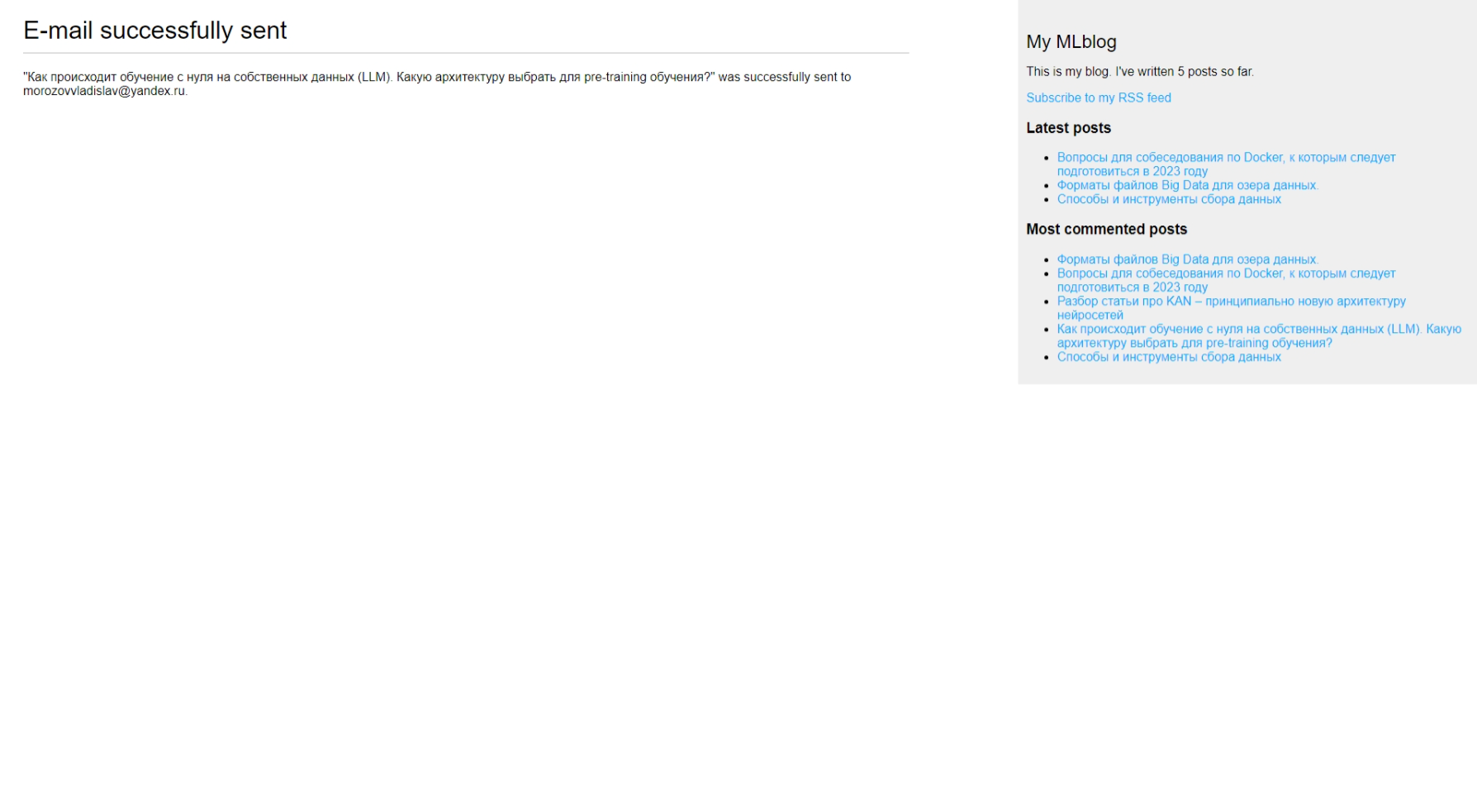


### Список постов

### Детальная информация о посте

### Сообщение о добавленном комментарии

### Новостная лента



### Сообщение о удачной отправке рекомендации

### Рекомендация, полученная по электронной почте