Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 804 «Теории вероятностей и математического моделирования»
Дисциплина: «Web-технологии разработки прикладных программных систем»

Курсовая работа

Студент: Цыкин И. А.
Группа: М8О-104М-23
Преподаватель: Азанов В.М.
Дата: 27.12.2023
Оценка:

Москва, 2023

Введение

Django – это мощный Python-фреймворк для разработки веб-приложений. Джанго предоставляет разработчикам огромный выбор готовых модулей, надстроек и инструментов, которые значительно ускоряют и упрощают процесс создания сложных, многофункциональных веб-приложений.

Фреймворк написан на Python и вся бэкенд-логика при разработке приложений тоже пишется на Python. Однако в шаблонах, как мы увидим позже, используются HTML, CSS, а при необходимости – JavaScript и его фреймворки.

Фреймворк славится своими «батарейками» – это те самые готовые модули и надстройки, которые сильно упрощают работу. В Django таких модулей много – панель администрирования, аутентификация и авторизация, формы с автоматической валидацией данных, встроенный сервер и так далее. Среди других плюсов фреймворка:

- **Универсальность**. Существует огромное количество дополнительных модулей, которые позволяют реализовать на Django приложение с любой функциональностью и для любой сферы.
- **ORM** – удобная прослойка для работы с базой данных, которая избавляет разработчика от необходимости писать запросы на языке SQL. Мы подробно разберем ORM в главе, посвященной работе с базами данных.
- **Оптимизация производительности.** Встроенные механизмы кэширования, обработка статических файлов и оптимизация запросов позволяют настроить Django приложение на максимальную производительность.
- **Простота интеграции.** Фреймворк без труда можно интегрировать с чем угодно – от блога на WordPress до GraphQL. Фронтенд для Django приложения можно сделать на любом JavaScript фреймворке – передачу данных с бэкенда на фронт и обратно можно реализовать даже без API. В бэкенде многих высоконагруженных платформ Python и Django используются совместно с другими языками программирования.
- **Гибкость.** На Django можно делать фуллстек-приложения, в которых вывод данных и все манипуляции с ними на фронтенде обеспечиваются средствами самого фреймворка. Но можно делать и гибридные приложения, в которых обработка данных разделена между некоторыми модулями Django и JavaScript фронтендом. А еще можно делать на Django (и Django REST Framework) только бэкенд и API, предоставляя всю свободу действий фронтенду.
- **Безопасность.** Фреймворк имеет надежную встроенную защиту от внедрения SQL-кода, подделки межсайтовых запросов и межсайтового скриптинга.

- **Скорость разработки.** Процесс разработки приложений на этом фреймворке занимает меньше времени, чем при использовании других платформ – благодаря первоклассной документации и огромному количеству готовых модулей.

Для избежания ситуации с обновлением или ошибочной установки библиотек для реализации блога необходимо создать виртуальную среду, где будут установлены необходимые работающие библиотеки.

Создание проекта Blog на Django

Первый шаг — создать новый проект Django. Попросту, это значит что мы запустим несколько стандартных скриптов из поставки Django, который создадут для нас скелет проекта. Это просто куча каталогов и файлов, которые мы будем использовать позже.

В Windows:

```
(venv) django-admin startproject mysite .
```

django-admin.py это скрипт, который создаст необходимую структуру директорий и файлы для нас. Ты должна теперь иметь следующую структуру проекта:

```
HalloDjango
├──────manage.py
└──────mysite
        settings.py
        urls.py
        wsgi.py
        __init__.py
```

manage.py это другой скрипт, который помогает с управлением сайтом. С помощью него мы сможем запустить веб-сервер на твоем компьютере без установки дополнительных программ.
Файл settings.py содержит настройки для твоего веб-сайта.
Файл urls.py содержит список шаблонов, по которым ориентируется urlresolver.

Настройка базы данных

Существует множество различных баз данных, которые могут хранить данные для твоего сайта. Мы будем использовать стандартную -- sqlite3.
Она уже выбрана по умолчанию в файле mysite/settings.py:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Файл запуска блога manage.py

```
import os
import sys


def main():
    """Run administrative tasks."""
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'HelloDjango.settings')
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)


if __name__ == '__main__':
    main()
```

```
(env) C:\Users\itsyk\python\WEB project\HelloDjango>py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
January 08, 2024 - 19:04:05
Django version 4.2.7, using settings 'HelloDjango.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

При переходе по ссылке http://127.0.0.1:8000/ открывается блог.

Models.py

```
from django.db import models
from django.conf import settings
from django.urls import reverse
from django.utils import timezone
from .managers import PostPublishedManager, PostManager
from django.template.defaultfilters import truncatewords


# Create your models here.
class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=200, verbose_name="Заголовок")
    text = models.TextField(verbose_name="Текст статьи")
    created_data  =  models.DateTimeField(default=timezone.now,  verbose_name="Дата
создания")
    published_date = models.DateTimeField(blank=True, null=True, verbose_name="Дата
публикации")
    is_published    =    models.BooleanField(default=False,    verbose_name="Запись
опубликована?")
    objects = PostManager()
    published = PostPublishedManager()

    def get_text_preview(self):
        return truncatewords(self.text, 10)
```

```
    def is_publish(self):
       return True if self.published_date else False

    def publish(self):
        self.published_date = timezone.now()
        self.is_published = True
        self.save()

    class Meta:
        verbose_name = 'Запись в блоге'
        verbose_name_plural = 'Запись в блоге'

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        """
        Возвращает URL для просмотра полного текста поста.
        """
        return reverse('blog:post_detail', args=[str(self.id)])

class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')
    author = models.CharField(max_length=200)
    text = models.TextField(verbose_name="Комментарий")
    created_data  =  models.DateTimeField(default=timezone.now,  verbose_name="Дата
создания")
    approved_comment = models.BooleanField(default=False, verbose_name="Одобрен?")

    def approve(self):
        self.approved_comment = True
        self.save()

    def __str__(self):
        return self.text

    class Meta:
        verbose_name = 'Комментарий'
        verbose_name_plural = 'Комментарии'
```

Managers.py

```
from django.db import models
from django.db.models import QuerySet, Manager
from django.utils import timezone

class PostPublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(
            published_date__lte=timezone.now())

class PostQuerySet(QuerySet):
    def for_user(self, user=None):
        if user.is_staff:
            return self.all()
        else:
            return self.filter(published_date__lte=timezone.now())

class PostManager(Manager):
    def get_queryset(self):
        return PostQuerySet(self.model, using=self._db)
```

```
    def for_user(self, user=None):
        return self.get_queryset().for_user(user=user)
```

Forms.py

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


class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ('title', 'text', 'is_published')

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

Serializers.py

```
from rest_framework import serializers, viewsets
from blog.models import Comment, Post

class CommentSerializer(serializers.ModelSerializer):
       text = serializers.CharField(max_length=200)
       created_data = serializers.DateTimeField()

       class Meta:
              model = Comment
              fields = ('text', 'created_data')

class BlogPostListSerializer(serializers.ModelSerializer):
       preview_text =serializers.SerializerMethodField()

       def get_preview_text(self, post):
              return post.get_text_preview()

       class Meta:
              model = Post
              fields = ('title', 'author', 'created_data', 'preview_text')

class BlogPostCreateUpdateSerialer(serializers.ModelSerializer):
       class Meta:
              model = Post
              exclude = ()

class BlogPostDetailSerializer(serializers.ModelSerializer):
       comments = CommentSerializer(many=True, read_only=True)
       comments_count = serializers.SerializerMethodField()

       def get_comments_count(self, obj):
              #return Comment.objects.filter(post=obj)
              return obj.comments.count()

       class Meta:
              model = Post
              fields = ('title', 'author', 'created_data', 'preview_text', 'comments',
'comments_count')
```

Views.py

```
from django.http import Http404
from django.shortcuts import render, get_object_or_404, redirect
from django.contrib.auth.decorators import login_required
from django.utils import timezone
from .forms import PostForm, CommentForm
from .models import Post, Comment
from .models import Comment

# Create your views here.
def post_list(request):
    posts = Post.objects.for_user(user=request.user)
    return render(request, 'blog/post_list.html', {'posts': posts})

def post_detail(request, id):
    post = get_object_or_404(Post, id=id)
    if not post.is_publish() and not request.user.is_staff:
        raise Http404("Запись в блоге не найдена")
    return render(request, 'blog/post_detail.html', {'post': post})

def handler404(request, exception, template_name="404.html"):
    response = render(request, "404.html")
    response.status_code = 404
    return response

def post_add(request):
   if request.user.is_authenticated:
       if request.method == "POST":
           form = PostForm(request.POST)
           if form.is_valid():
               post = form.save(commit=False)
               post.author = request.user
               post.published_date = timezone.now()
               post.save()
               return redirect('blog:post_detail', id=post.id)
       else:
               form = PostForm()
       return render(request, 'blog/post_edit.html', {'form': form})
   return redirect('post_list')

@login_required
def post_edit(request, id):
    post = get_object_or_404(Post, id=id) if id else None

    if post and post.author != request.user:
        return redirect('post_list')

    if request.method == "POST":
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            if post.is_published:
                post.published_date = timezone.now()
            else:
                post.published_date = None
            post.save()
            return redirect('blog:post_detail', id=post.id)
    else:
        form = PostForm(instance=post)
    return render(request, 'blog/post_edit.html', {'form': form})\
```

```
@login_required
def post_publish(request, id):
    post = get_object_or_404(Post, id=id)
    post.publish()
    return redirect('blog:post_detail', id=id)

def add_comment(request, id):
    post = get_object_or_404(Post, id=id)
    if request.method == "POST":
        form = CommentForm(request.POST)
        if form.is_valid():
            comment = form.save(commit=False)
            comment.post = post
            comment.created_date = timezone.now()
            comment.author = request.user
            comment.save()
            return redirect('blog:post_detail', id = post.id)
    else:
        form = CommentForm()
    return render(request, 'blog/add_comment.html', {'form': form})
```

Urls.py

```
from django.urls import include, path
from . import views
from rest_framework import routers

from core.views import CommentViewSet, BlogPostViewSet

app_name = 'blog'
router = routers.DefaultRouter()
router.register(r'comments', CommentViewSet)
router.register(r'posts', BlogPostViewSet)


urlpatterns = [
       path('API', include(router.urls)),
    path('', views.post_list, name="post_list"),
    path('posts/<int:id>/', views.post_detail, name='post_detail'),
    path('posts/<int:id>/edit/', views.post_edit, name='post_edit'),
    path('posts/<int:id>/publish/', views.post_publish, name='post_publish'),
    path('post/add/', views.post_add, name='post_add'),
    path('comment/<int:id>/add/', views.add_comment, name='add_comment'),
]
```

Для вывода результата страницы реализованы различные файл формата .html и .css.

Test_models.py

```
from django.contrib.auth import get_user_model
from django.test import TestCase
from django.utils import timezone

from ..models import Post

User = get_user_model()


class PostTest(TestCase):
```

```
	def setUp(self):
		author1 = User.objects.create(username='author #1')
		author2 = User.objects.create(username='author #2')
		Post.objects.create(title='Test Post #1', text='Dummy text #1',
author=author1)
		Post.objects.create(title='Test Post #2', text='Dummy text #2',
author=author2)
		Post.objects.create(title='Test Post #3', text='Dummy text #3',
is_published=True, author=author2)

	def test_published_method_for_post(self):
		post = Post.objects.get(title='Test Post #1')
		post.publish()
		self.assertEqual(post.is_published, True)

	def test_published_post_filtering(self):
		post = Post.objects.get(title='Test Post #1')
		post.publish()
		posts = Post.published.all()
		self.assertEqual(posts.count(), 1)
```

Test_views.py

```
from django.contrib.auth import get_user_model
from django.test import TestCase, Client
from django.urls import reverse
from rest_framework import status
from rest_framework.test import APIClient
import json

from ..models import Post
from ..serializers import BlogPostListSerializer, BlogPostDetailSerializer

User = get_user_model()
client = Client()

class GetAllPostsTest(TestCase):
	def setUp(self):
		author = User.objects.create(username='author #1')
		Post.objects.create(title='Test Post #1', text='Dummy text #1',
author=author)
		Post.objects.create(title='Test Post #2', text='Dummy text #2',
author=author)
		Post.objects.create(title='Test Post #3', text='Dummy text #3',
author=author)

	def test_get_all_posts(self):
		response = client.get(reverse('blog:post-list'))
		posts = Post.objects.all()
		serializer = BlogPostListSerializer(posts, many=True)
		self.assertEqual(response.data, serializer.data)
		self.assertEqual(response.status_code, status.HTTP_200_OK)

class GetSinglePostTest(TestCase):
	def setUp(self):
		author = User.objects.create(username='test')
		self.post = Post.objects.create(title='Test Post #1', author=author)

	'''def test_get_valid_single_post(self):
				response = client.get(reverse('blog:post-detail',
kwargs={'pk': self.post.pk}))
```

```
                                post = Post.objects.get(pk=self.post.pk)
                                serializer = BlogPostDetailSerializer(post)
                                self.assertEqual(response.data, serializer.data)
                                self.assertEqual(response.status_code,
status.HTTP_200_OK)'''

      def test_get_invalid_single_post(self):
            response = client.get(reverse('blog:post-detail', kwargs={'pk': 9999}))
            self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)

class CreateNewPostTest(TestCase):
      def setUp(self):
            self.author = User.objects.create(username='test')
            self.valid_payload = {
                  'title': 'Blog Post #1',
                  'text': 'Blog Post Description',
                  'author': 1,
            }
            self.invalid_payload = {
                  'title': 'Blog Post #1',
                  'author': 2,
            }

      def test_create_valid_single_post(self):
            response            =            client.post(reverse('blog:post-list'),
data=json.dumps(self.valid_payload), content_type='application/json')
            self.assertEqual(response.status_code, status.HTTP_201_CREATED)

      def test_create_invalid_single_post(self):
            response            =            client.post(reverse('blog:post-list'),
data=json.dumps(self.invalid_payload), content_type='application/json')
            self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)

class UpdateSinglePostTest(TestCase):
      def setUp(self):
            self.author = User.objects.create(username='test')
            self.post   =   Post.objects.create(title='Blog   Post   #1',   text='Post
Description', author=self.author)
            self.valid_payload = {
                  'title': 'Blog Post #1',
                  'text': 'Blog Post Description',
                  'author': 1,
            }
            self.invalid_payload = {
                  'title': 'Blog Post #1',
                  'test': None,
                  'author': 2,
            }

      def test_valid_update_post(self):
            response    =    client.put(reverse('blog:post-detail',    kwargs={'pk':
self.post.pk}), data=json.dumps(self.valid_payload), content_type='application/json')
            self.assertEqual(response.status_code, status.HTTP_200_OK)

      def test_invalid_update_post(self):
            response    =    client.put(reverse('blog:post-detail',    kwargs={'pk':
self.post.pk}),                                data=json.dumps(self.invalid_payload),
content_type='application/json')
            self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)

class DeleteSinglePostTest(TestCase):
      def setUp(self):
```

```
            self.author = User.objects.create(username='test')
            self.post = Post.objects.create(title='Blog Post #1', text='Post
Description', author=self.author)

      def test_valid_delete_post(self):
            response = client.delete(reverse('blog:post-detail', kwargs={'pk':
self.post.pk}))
            self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)

      def test_invalid_delete_post(self):
            response = client.delete(reverse('blog:post-detail', kwargs={'pk':
9999}))
            self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)

class PublishSinglePostTest(TestCase):
      def setUp(self):
            self.api = APIClient()
            self.author = User.objects.create(username='test', password='test')
            self.post = Post.objects.create(title='Blog Post #1', text='Post
Description', author=self.author, is_published=False)

      def test_unauth_publish_post(self):
            response = client.post(reverse('blog:post-publish'),
args=[self.post.pk])
            self.assertEqual(response.status_code, status.HTTO_403_FORBIDDEN)

      def test_autheticated_publish_post(self):
            self.api.force_authenticate(user=self.author)
            response = self.api.post(reverse('blog:post-publish'),
args=[self.post.pk])
            self.assertEqual(response.status_code, status.HTTP_200_OK)
            post = Post.objects.get(titlr='Blog Post #1')
            self.assertEqual(post.is_published, True)
```

Unit тесты

Во-первых, это способ проверить работу нового функционала, который мы пишем. Также в процессе развития программного продукта код периодически нужно рефакторить. В этом случае юнит-тесты позволяют проводить рефакторинг без опасений, что существующая функциональность будет сломана. Помимо этого, юнит-тесты позволяют облегчить обнаружение ошибок и локализовать их поиск. Другое полезное свойство юнит-тестов в том, что, читая их, можно понять, как работают части системы и как их нужно использовать. Это документация, которая живет и меняется вместе с кодом. Из 13 тестов запустились 10.

```
(env) C:\Users\itsyk\python\WEB project\HelloDjango>py manage.py test blog
Found 10 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..........
----------------------------------------------------------------------
Ran 10 tests in 0.024s

OK
Destroying test database for alias 'default'...
```

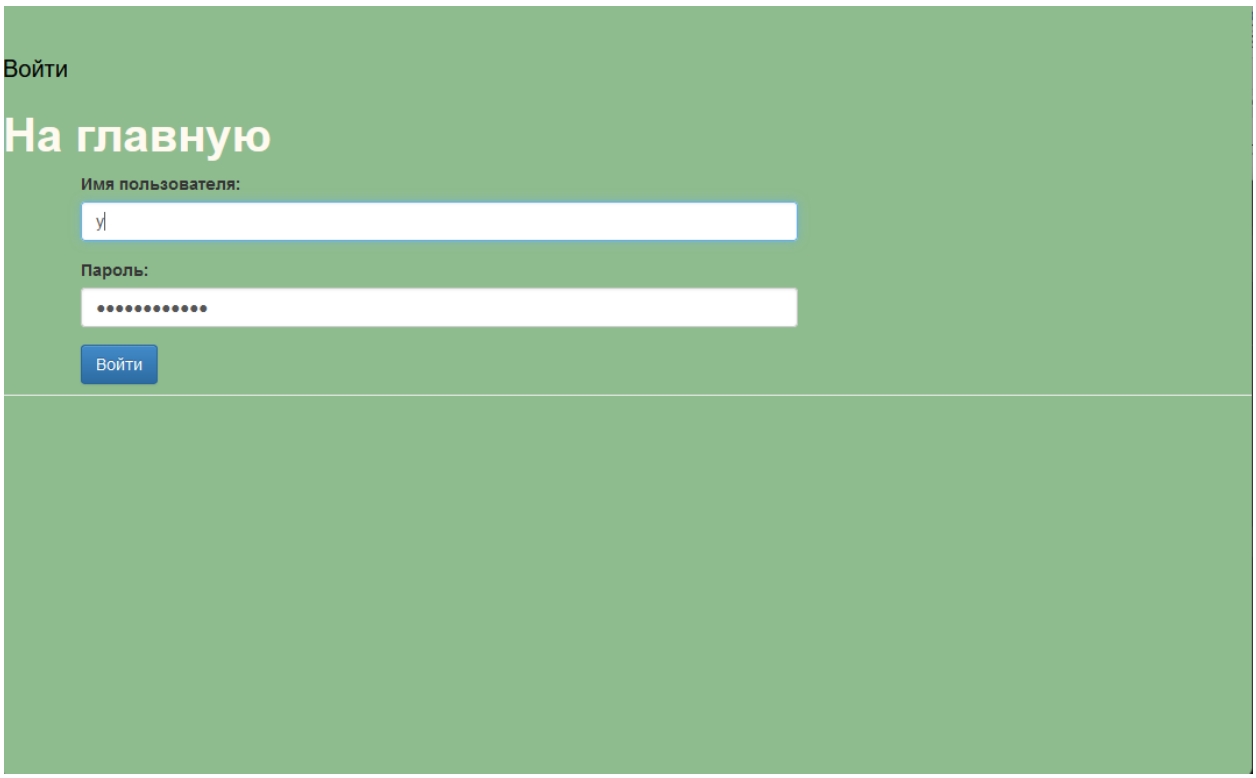

Рисунок 1 авторизация в блоге

Рисунок 2 Основная страница

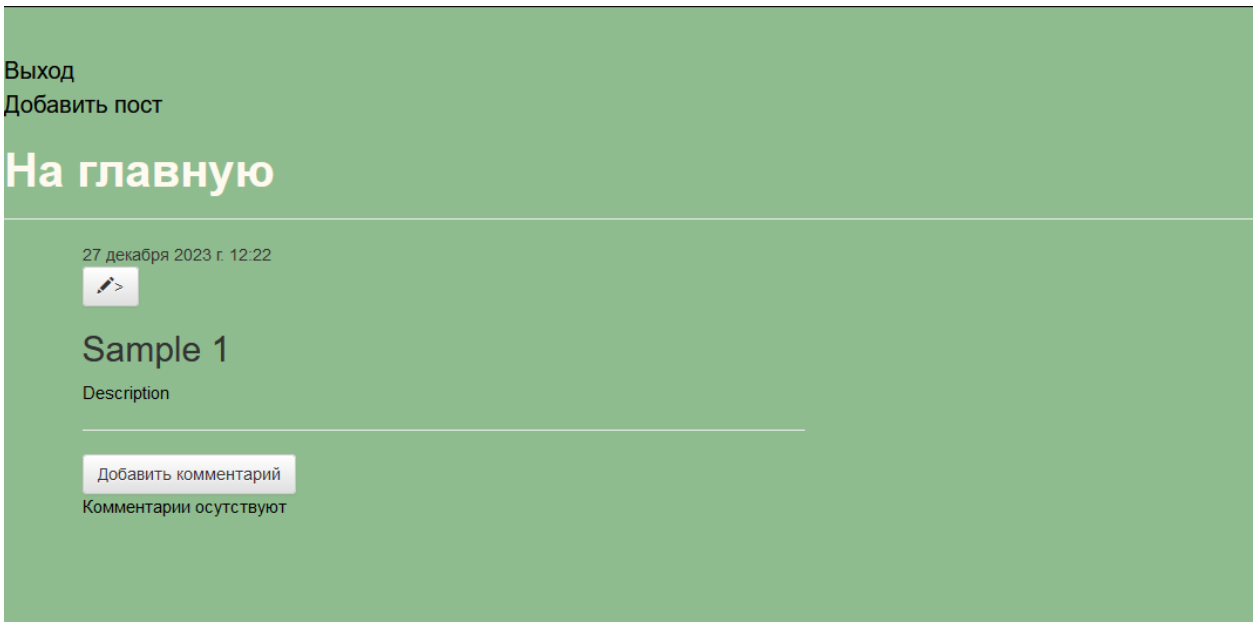

Рисунок 3 Отдельный пост

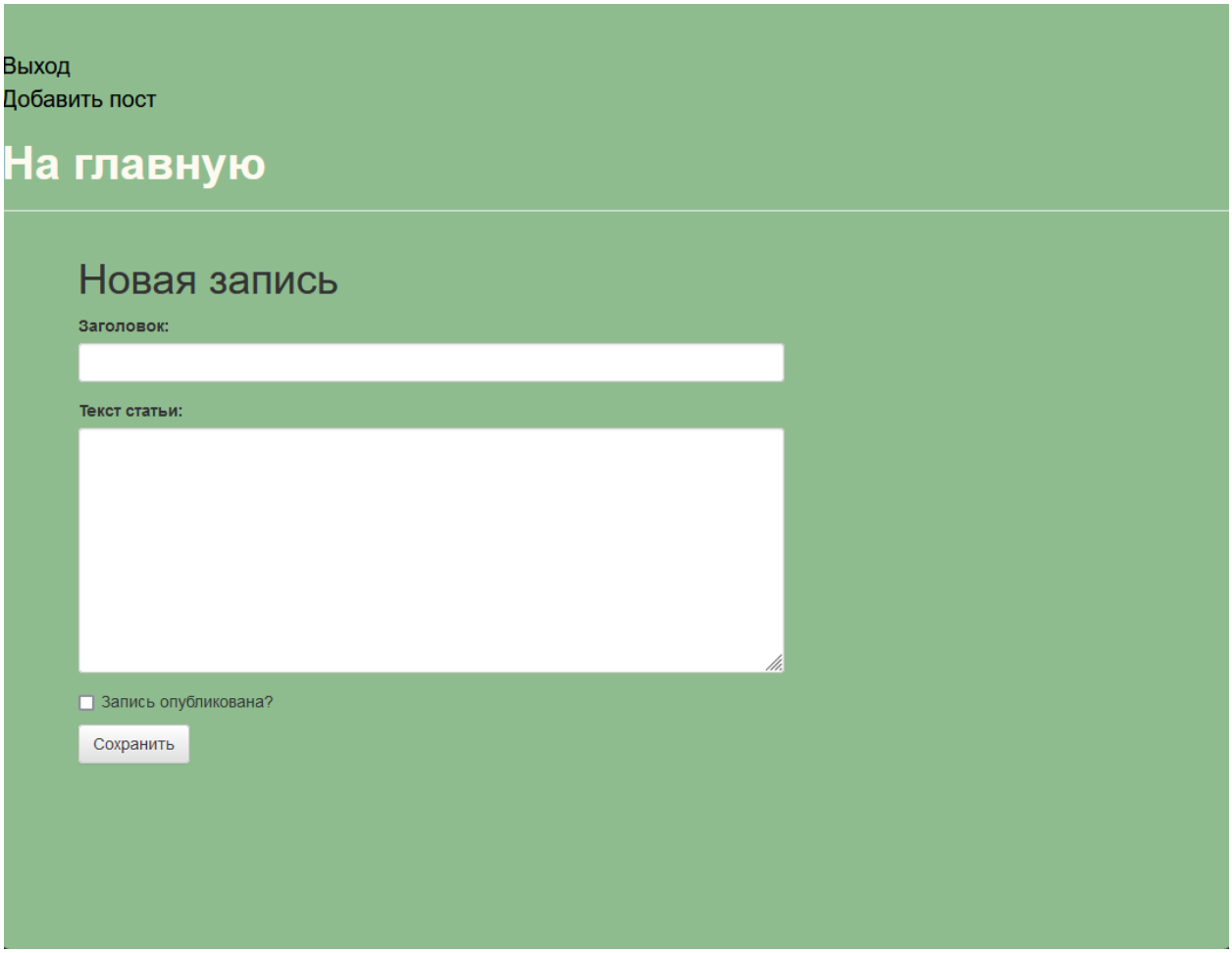

Рисунок 4 Добавление поста

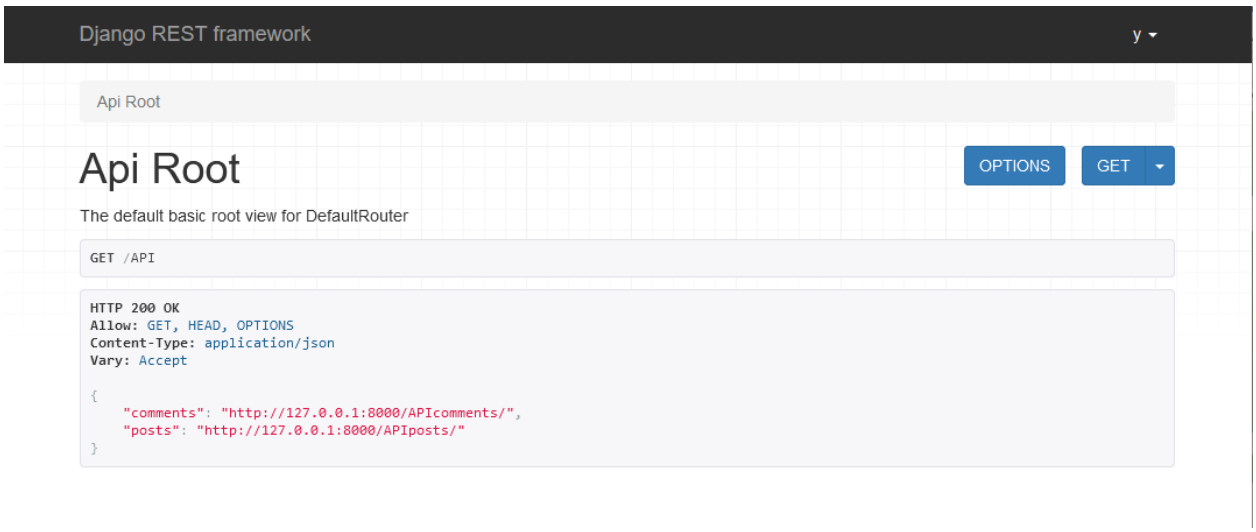

Рисунок 5 API