

Министерство цифрового развития, связи и массовых коммуникаций  
Российской Федерации  
Ордена Трудового Красного Знамени  
федеральное государственное бюджетное  
образовательное учреждение высшего образования  
«Московский технический университет связи и информатики»

Кафедра МКиИТ  
Проектирование клиент-серверных приложений

Лабораторная работа №5  
“Создание формы и представления для нового поста”

Выполнил:  
студент 3 курса,  
группы БФИ2001  
Лушин Е. А.

Москва 2023

**Цель работы:** научиться создавать формы и представления для нового поста.

**Задание:**

Создание формы и представления для нового поста

- Создайте стили, подключив css-файл к шаблону, которые соответствуют макету **lab5\_creation\_form**.
- Добавьте проверку на то, что введенное для нового поста название уникально.

## Краткая теория

### Работа с формами в представлениях Django

HTML-формы — становой хребет интерактивных веб-сайтов. Это может быть единственное поле для ввода поискового запроса, как на сайте Google, вездесущая форма для добавления комментария в блог или сложный специализированный интерфейс ввода данных. В этой главе будет рассказываться, как Django позволяет обратиться к данным, которые отправил пользователь, проверить их и что-то с ними сделать. Попутно пойдет речь об объектах `HttpRequest`.

Как вы помните, любая функция представления принимает объект `HttpRequest` в качестве первого параметра.

Пример функции-представления `hello`:

```
from django.http import HttpResponse  
def hello(request):  
    return HttpResponse("Hello world")
```

У объекта с типом данных `HttpRequest`, каковым является переменная `request`, есть целый ряд интересных атрибутов и методов, с которыми необходимо познакомиться, чтобы знать, какие существуют возможности. С их помощью можно получить информацию о текущем запросе (например, имя пользователя или версию браузера, который загружает страницу вашего сайта) в момент выполнения функции представления.

Например, у объекта `request` есть атрибут `META` (доступ к нему можно получить так: `request.META`) — это словарь Python, содержащий все HTTP-заголовки данного запроса, включая IP-адрес пользователя и информацию об агенте пользователя (обычно название и номер версии веб-браузера). В список входят как заголовки, отправленные пользователем, так и те, что были установлены вашим веб-сервером. Ниже перечислены некоторые часто встречающиеся ключи словаря:

- `HTTP REFERER`: ссылающийся URL, если указан. (Обратите внимание на ошибку в написании слова `REFERER`.)
- `HTTP USER AGENT`: строка с описанием агента пользователя (если указана). Выглядит примерно так: «Mozilla 5.0 (X11; U; Linux i686) Gecko/20080829 Firefox/2.0.0.17».
- `REMOTE ADDR`: IP-адрес клиента, например «12.345.67.89». (Если запрос проходил через прокси-серверы, то это может быть список IP-адресов, разделенных запятыми, например «12.345.67.89,23.456.78.90».

Отметим, что поскольку `request.META` — обычный словарь Python, то при попытке обратиться к несуществующему ключу будет возбуждено исключение `KeyError`. (Ведь HTTP-заголовки — это внешние данные, то есть отправлены пользовательским браузером, доверять им нельзя, поэтому вы должны проектировать свое приложение так, чтобы оно корректно обрабатывало ситуацию, когда некоторый заголовок пуст или отсутствует.) Нужно либо использовать `try/except`-блок, либо осуществлять доступ по ключу методом `get()`.

Пример работы со словарем `request.META`:

*# ПЛОХО!*

*def ua\_display\_bad(request):*

*ua = request.META['HTTP USER AGENT']*

*# Может возникнуть KeyError!*

*return HttpResponse("Ваш браузер %s" % ua)*

*# ХОРОШО (ВАРИАНТ 1)*

```

def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Ваш браузер %s" % ua)
# ХОРОШО (ВАРИАНТ 2)
def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Ваш браузер %s" % ua)

```

Помимо основных метаданных о запросе объект `HttpRequest` имеет два атрибута, в которых хранится отправленная пользователем информация: `request.GET` и `request.POST`. Тот и другой объекты похожи на словарь и дают доступ к данным, отправленным соответственно методом GET или POST. POST-данные обычно поступают из HTML-формы (тег `<form>`), а GET-данные — из формы или строки запроса, указанной в гиперссылке на странице.

В общем случае, с точки зрения разработки, у формы есть две стороны: пользовательский HTML-интерфейс и код для обработки отправленных данных на стороне сервера. С первой частью все просто, вот представление для отображения формы поиска.

Пример представления, отображающего форму:

```

from django.shortcuts import render

def search(request):
    return render(request, 'search form.html', {})

```

Представление можно поместить в файл `application name/views.py`.

Соответствующий шаблон `search form.html` мог бы выглядеть так (пример простой формы вопроса):

```

<html>
<head>
<title> Поиск </title>

```

```
</head>
<body>
    <form method="POST">
        <input type="text" name="question">
        <input type="submit" value="Найти">
    </form>
</body>
</html>
```

А вот образец URL в файле `urls.py` (пример конфигурации URL для страницы с формой):

```
from mysite.books.views import search
urlpatterns = patterns('
#...
(r'^search-form/$', search),
#...
)
```

Если теперь запустить сервер разработки и зайти на страницу по адресу `http://127.0.0.1:8000/search-form/`, то появится интерфейс поиска. Все просто.

Однако, заполнив поле ввода в форме и нажав кнопку «Найти», единственное, что вы увидите. . . , та же самая форма, только уже опустевшая. Почему такое произошло? Потому что в данном случае, когда вы нажали кнопку «Найти», был отправлен запрос на тот же адрес URL (обратите внимание, что в адресной строке браузера ничего не поменялось), по этому адресу у вас в конфигурации URL поставлена та же функция представления, которая просто занимается рендерингом шаблона. Но раз при отправлении данных в форму происходит запуск представления `search`, то, значит, в нем и надо как-то обработать поступающие данные. Осталось только разобраться, а что пользователю сейчас от сервера надо: просто получить страницу с формой (когда пользователь первый раз зашел на сайт) или обработать данные с уже заполненной формы (то есть пользователь минимум второй раз на странице).

Чтобы отличить одно от другого, можно проверить метод, которым отправлялись данные. Метод «GET» по стандарту используется только для получения какой-либо информации с сервера (в данном случае информацией является html-код возвращаемой страницы), методом же «POST» отправляются запросы, которые должны сохранить на сервере какую-либо информацию (например, когда пользователь отправляет свой логин-пароль для регистрации). Соответственно, здесь в случае метода «GET» нужно просто отрендерить страницу с формой и вернуть её, а если использовался метод «POST», то нужно с помощью переменной `request` прочитать введенные данные и исходя из них вернуть какой-то определенный ответ или просто перенаправить на другую страницу. Для проверки можно просто сравнить содержимое атрибута `request.method`, в котором содержится название используемого метода (пример проверки метода запроса):

```
if request.method == "POST":  
    # обработать данные формы, если метод POST  
else:  
    # просто вернуть страницу с формой, если метод GET  
return render(request, 'create post.html', {})
```

Чтобы получить доступ к самим данным, следует использовать объект, напоминающий словарь, `request.POST`. В нем содержатся все данные, которые были переданы в форме.

Пример получения доступа к данным, переданным через форму:

```
if request.method == "POST":  
    # обработать данные формы, если метод POST  
    question = request.POST['question']  
else:  
    # просто вернуть страницу с формой, если метод GET  
    return render(request, 'create post.html', {})
```

Обратите внимание, что у словаря request.POST имеется ключ question. Название ключа берется из имени инпута, который находится в отправленной форме. Помните шаблон формы, там как раз поле ввода имело следующий код:

```
<input type="text" name="question">
```

Если бы вместо name="question" было указано что-то другое, например name="interest", то и в функции-представлении пришлось бы обращаться по ключу interest.

Попробуйте запустить тестовый сервер и создать функцию-представление, которая будет спрашивать ваше имя и, если оно введено, будет возвращать приветствие по этому имени. Шаблон с именем "greetings.html" для этой цели можете использовать следующий:

```
<html>
<head>
  <title> Приветствие </title>
</head>
<body>
  <h1> Введите ваше имя </h1>
  <form method="POST">
    <input type="text" name="first name">
    <input type="submit" value="Найти">
  </form>
  {% if first name %}
  Привет, {{first name}}!
  {% endif %}
</body>
</html>
```

Тогда файл views.py должен содержать примерно такую функцию (пример представления, возвращающего шаблон с приветствием):

```
from django.shortcuts import render
def search(request):
```

```

if request.method == "POST":
    # обработать данные формы, если метод POST
    first name = request.POST['first name']
    # вернуть страницу с приветствием, если метод GET
    return render(
        request,
        'greetings.html',
        {'first name': first name}
    )
else:
    # просто вернуть страницу с формой, если метод GET
    return render(request, 'greetings.html', {})

```

Однако у такого подхода есть существенный минус, который упоминался ранее: если пользователь воспользуется методом POST, но не отправит поле имени или это поле переименует, будет возбуждена ошибка `KeyError`, потому что в строке (пример неправильной работы со словарем `request.POST`):

```
first name = request.POST['first name']
```

будет обращение к ключу словаря, которого не существует. Хорошей практикой, как вы, конечно, помните, является использование метода `get()` у словарей, который позволяет безопасно извлекать значения по ключам и указывать значения по умолчанию вторым параметром (пример правильной работы со словарем `request.POST`):

```
first name = request.POST.get('first name', 'Anonymous')
```

В данном случае, если ключа «first name» в словаре не окажется, вместо реального имени будет использоваться строка «Anonymous».



## Выполнение

За основу был взят проект из лабораторной работы №4. В файле `urls.py` создал ещё один адрес: `article/new`.

Листинг 1. Содержимое файла `urls.py`

```
blog > blog >  urls.py > ...
1  from articles import views
2  from django.contrib import admin
3  from django.urls import path
4
5  urlpatterns = [
6      path('admin/', admin.site.urls),
7      path('', views.archive, name= 'home'),
8      path('article/<int:article_id>', views.get_article, name= 'get_article'),
9      path('article/new/', views.create_post, name= 'create_post'),
10 ]
```

Данный адрес будет обрабатываться функцией-представлением `create_post`. Эта функция должна будет при первом обращении к странице просто возвращать `html`-код с формой создания новой записи, а в случае, если к функции пользователь обратился во второй и следующий разы, при этом заполнив форму с названием и текстом нового поста, представление должно сохранять отправленные через форму данные в базу данных.

В представлении должна быть проверка на то, авторизован ли пользователь или нет. Если пользователь авторизован, то сначала программе необходимо понять, что нужно сделать на данный момент: просто вернуть страницу с формой или пользователь в свое время уже получил форму, заполнил её и сейчас от программы требуется сохранить новую запись. Для этого нужно знать метод, с помощью которого отправлен запрос, и если это метод `POST`, то необходимо проверить полученные данные (если не заполнено одно из полей, вернуть обратно страницу с формой и уведомить об ошибке) и, если проверка пройдена, создать в базе данных новую запись. Также согласно заданию, в конце методических указаний к лабораторной работе, необходимо создать проверку на то, является ли название статьи уникальным.

## Листинг 2. Содержимое файла views.py

```
blog > articles > views.py > ...
1  from .models import Article
2  from django.shortcuts import render
3  from django.http import Http404
4
5  # Create your views here.
6
7  def archive(request):
8      return render(request, 'archive.html', {"posts": Article.objects.all()})
9
10 def get_article(request, article_id):
11     try:
12         post = Article.objects.get(id=article_id)
13         return render(request, 'article.html', {"post": post})
14     except Article.DoesNotExist:
15         raise Http404
16
17 def create_post(request):
18     if request.user.is_authenticated:
19         if request.method == "POST":
20             # обработать данные формы, если метод POST
21             form = {
22                 'text': request.POST["text"], 'title': request.POST["title"]
23             }
24             # в словаре form будет храниться информация, введенная пользователем
25             if form["text"] and form["title"]:
26                 # если поля заполнены без ошибок
27                 if not Article.objects.filter(title=form["title"]).exists():
28                     Article.objects.create(text=form["text"], title=form["title"], author=request.user)
29                     return redirect('get_article', article_id=Article.objects.count())
30                 else:
31                     form['errors'] = u"Статья с таким названием уже существует"
32                     return render(request, 'create_post.html', {'form': form})
33             # перейти на страницу поста
34             else:
35                 # если введенные данные некорректны
36                 form['errors'] = u"Не все поля заполнены"
37                 return render(request, 'create_post.html', {'form': form})
38         else:
39             # просто вернуть страницу с формой, если метод GET
40             return render(request, 'create_post.html', {})
41     else:
42         raise Http404
```

Далее необходимо создать HTML-документ для отображение новой страницы на нашем сайте для создания новой статьи.

## Листинг 2. Содержимое файла create\_post.html

```
blog > articles > templates > create_post.html > html
1  {% load static %}
2
3  !DOCTYPE html>
4  <html>
5
6  <head>
7      <title>Django Public Blog - создать статью</title>
8      <link rel="stylesheet" type="text/css" href="{% static 'css/article.css' %}">
9  </head>
10
11 <body>
12     <div class="content">
13         <h1>Написать статью</h1>
14         <form class="create" method="POST">{% csrf_token %}
15             <input type="text" name="title" placeholder="Название статьи" value="{{form.title}}" maxlength="100">
16             <textarea name="text" placeholder="Текст статьи">{{form.text}}</textarea>
17             <input type="submit" value="Сохранить" class="save_button">
18         </form>
19     </div>
20 </body>
```

Затем, создал и подключил стили к новой странице на нашем сайте. Часть стилей взял из предыдущей лабораторной работы, а часть написал сам для тегов «form», «input» и «textarea».

### Листинг 3. Содержимое файла article.css

```
body {
    background: #1abc9c;
    font-family: Tahoma, Arial, sans-serif;
    color: #ffffff
}

img {
    display: block;
    width: 318px;
    margin-left: auto;
    margin-right: auto;
}

.archive {
    width: 960px;
    margin-left: auto;
    margin-right: auto;
}

post-title a {
    color: #ffffff;
}

.article-author {
    width: 50%;
    float: Left;
}

.article-created-date {
    text-align: right;
}

.article-image {
    display: block;
    width: 318px;
    margin-left: 0;
}

.link {
    color: white;
    font-weight: bold;
    position: absolute;
    right: 470px;
    top: 180px;
```

```
}

.article-border p {
    text-align: right;
}

.article-text {
    width: 960px;
    text-align: justify;
}

.article-created-data {
    text-align: right;
}

.content {
    text-align: center;
    padding-top: 70px;
}

input[name="title"] {
    padding: 5px;
    margin-bottom: 10px;
    border: 1px solid #888;
    outline: none;
    -moz-appearance: none;
    width: 200px;
    text-align: center;
    border-radius: 40px;
}

textarea[name="text"] {
    padding: 25px;
    margin-bottom: 10px;
    border: 1px solid #888;
    outline: none;
    -moz-appearance: none;
    width: 650px;
    height: 350px;
    resize: none;
    border-radius: 40px;
    scrollbar-width: thin;
}

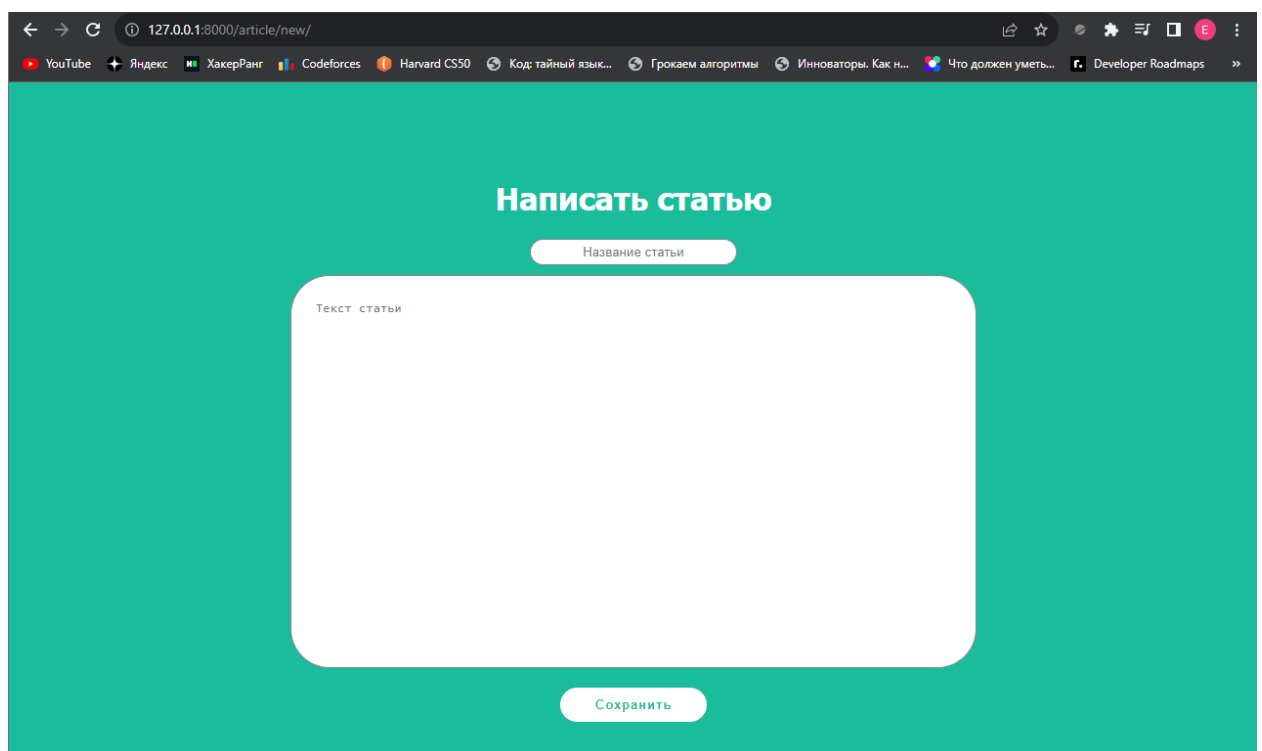
.create {
    display: flex;
    flex-direction: column;
    align-items: center;
}

.save_button {
```

```
padding: 10px;
width: 150px;
background-color: white;
border: none;
border-radius: 40px;
color: #1abc9c;
font-weight: bold;
letter-spacing: 0.06em;
margin-top: 10px;
}

.save_button:hover {
color: white;
background-color: #1abc9c;
box-shadow: 1px 1px 10px 10px;
transition-duration: 0.3s;
}
```

Далее представлены страница создания статьи и страница создания статьи с попыткой создания статьи с существующим названием на рисунках 1-2 на страницах 13 и 14 соответственно.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/article/new/'. The browser's bookmark bar includes links to YouTube, Яндекс, ХакеpPaнr, Codeforces, Harvard CS50, and several other educational and development-related sites. The main content area has a teal background and features a white form titled 'Написать статью' (Write an article). The form consists of a small input field for the title labeled 'Название статьи' and a large, rounded rectangular text area labeled 'Текст статьи'. At the bottom of the form is a white button labeled 'Сохранить' (Save).

Рисунок 1 – Страница создания статьи

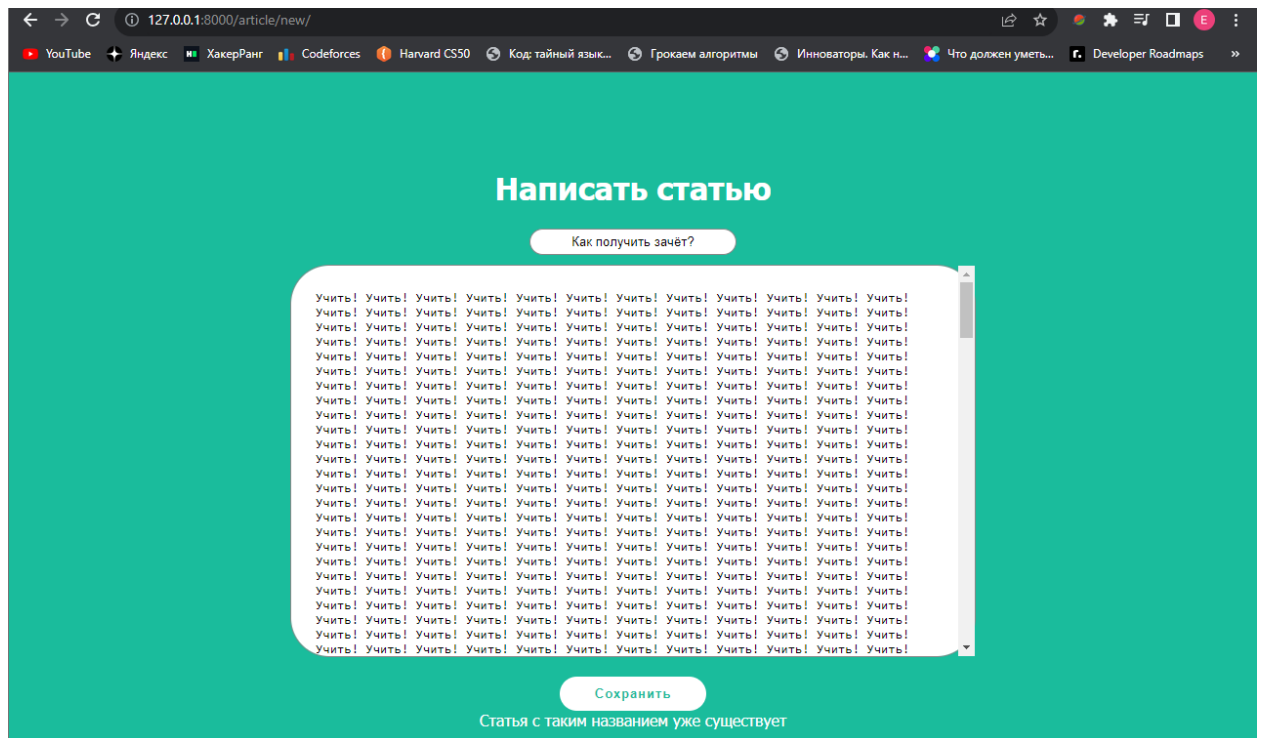


Рисунок 2 – Ошибка в случае попытки создания статьи с уже существующим названием

**Вывод:** в данной лабораторной работе я научился создавать страницу создания статьи для пользователей сайта.