Занятие #42. HTTP, GET, POST, параметры запросов, переменные в шаблонах, статические файлы

HTTP

HTTP расшифровывается как HyperText Transfer Protocol (протокол передачи гипертекста) и используется для передачи данных в сети Интернет.

HTTP основан на *клиент-серверной* технологии, в которой подразумевается существование двух взаимодействующих частей - *клиента* и *сервера*.

Клиент (англ. **client**) - программа или электронное устройство, которое отправляет *запросы* на сервер.

Сервер (англ. **server**, от англ. serve - служить, обслуживать) - программа, которая обрабатывает *запросы* от клиентов и возвращает результаты - *ответы*. Серверами также обычно называют компьютеры или виртуальные машины, на которых эти сервера запущены.

HTTP, как протокол, определяет, какие данные могут передаваться в запросах и ответах, в каком формате и порядке.

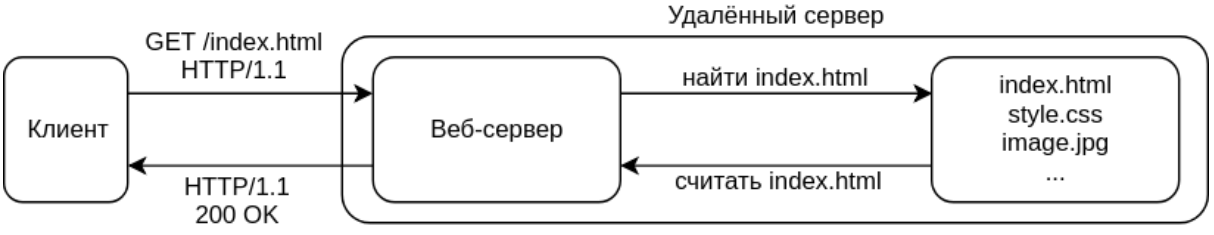
Например, в случае с тестовой страницей в Django сервером является приложение на питоне, запускаемое командой manage.py runserver, клиент - ваш браузер, а данные, приходящие в ответ на запрос - это содержимое html-файла.

Взаимодействие сервера и клиента

Чтобы отобразить веб-страницу, ваш браузер должен сделать **запрос** (англ. **request**) с информацией о том, что мы хотим от сервера. Если запрос корректен, сервер вернёт результат своей работы - **ответ** (англ. **response**). Этот процесс так и называют - **запрос-ответ** (англ. **request-response**).

В простом варианте клиент может запрашивать у сервера какой-либо *статический* файл, который уже хранится где-то на сервере, например index.html или style.css. Такие файлы называют **статическими**, т.к. они их содержимое постоянно и не меняется в зависимости от запроса.

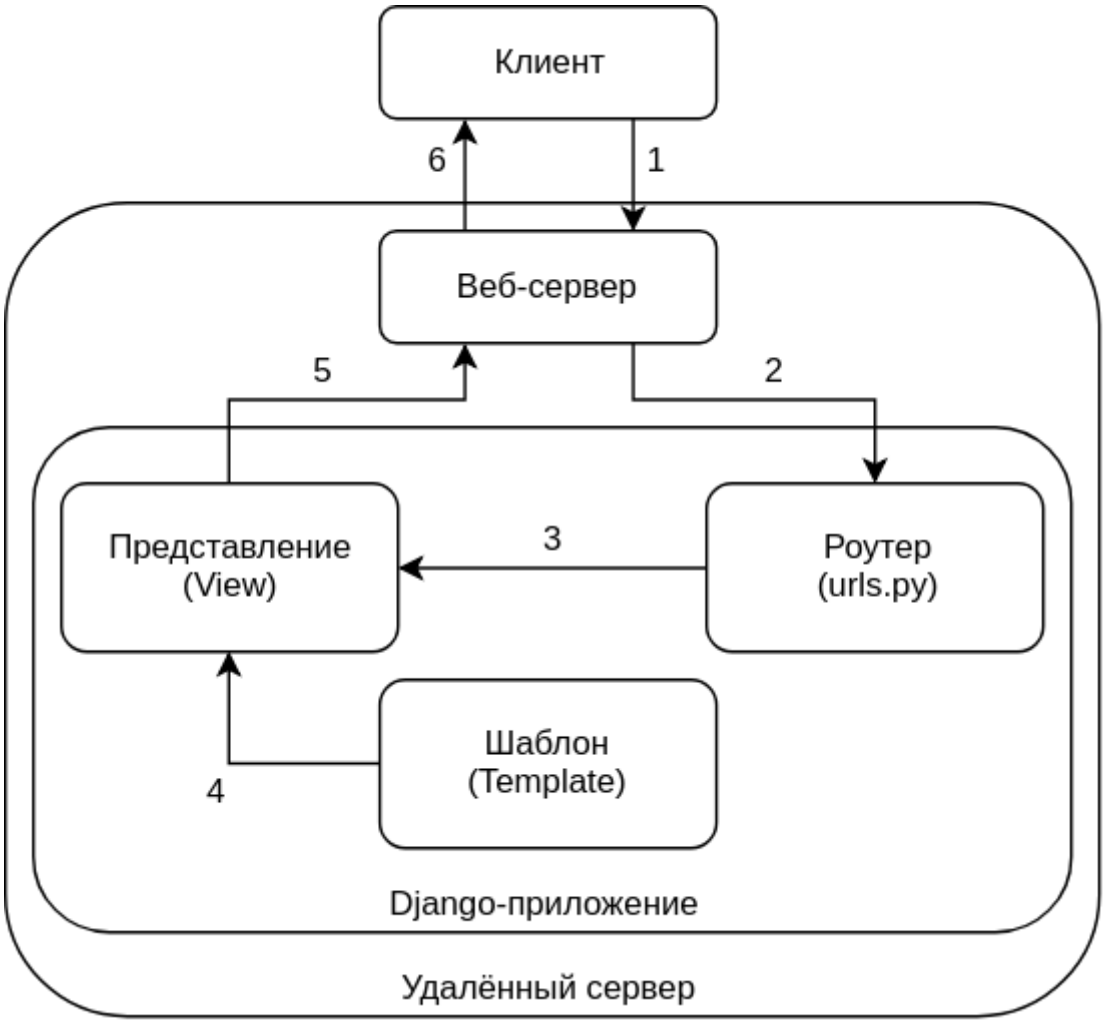
Общение между браузером (клиентом) и сервером в этом случае происходит по следующей схеме:



1. Клиент делает HTTP-запрос на сервер с указанием пути к файлу и его имени: /index.html.
2. Сервер считывает содержимое найденного файла.
3. Сервер возвращает содержимое файла в виде HTTP-ответа

Для каждого нового файла выполняется отдельный запрос. Например, если в файл index.html подключен файл стилей style.css, то браузер сделает дополнительный запрос для него. Если файл стилей использует картинку image.png - ещё один запрос. Всего 3 запроса на 3 файла.

В случае с приложением на Django (или другом подобном фреймворке) общение сервера и клиента происходит по другой схеме:



1. Клиент делает HTTP-запрос на сервер с указанием адреса страницы.
2. Сервер перенаправляет запрос в Django-приложение. Django-приложение сверяет адрес запроса с известными ему адресами в файле urls.py.
3. В зависимости от указанного адреса Django находит *представление*, которое будет обрабатывать запрос и передаёт запрос ему.
4. Представление *рендерит* ("рисует") веб-страницу в HTML на основе имеющегося в приложении *шаблона*.
5. Представление возвращает полученную HTML-страницу серверу.
6. Сервер помещает полученную страницу в HTTP-ответ и возвращает клиенту.

Такой подход позволяет на лету генерировать страницы с различной структурой и *динамически* подставлять в них разные данные в зависимости от параметров запроса и состояния приложения.

HTTP-запросы

HTTP-запросы бывают нескольких видов, или, как их называют, методов:

- GET - запрос на получение ресурса
- POST - запрос на создание ресурса
- PUT - запрос на изменение ресурса, полная замена
- PATCH - запрос на изменение ресурса, частичное обновление
- DELETE - запрос на удаление ресурса
- HEAD - аналогичен GET, но возвращает только заголовки
- OPTIONS - запрос описания параметров для соединения с ресурсом. Например, сервер может вернуть список разрешённых методов запроса по данному адресу.

Ресурсом в данном случае может быть какая-то запись в базе данных, файл или какой-либо другой объект, к которому можно обратиться подобным образом.

Например, если у нас на сайте есть статья по адресу `http://localhost:8000/articles/386`, то статья является ресурсом, её можно получить послав запрос GET, поменять её содержимое методами PUT или PATCH, и удалить методом DELETE.

На сайте могут быть и "общие" ресурсы, например - список статей: `http://localhost:8000/articles/`. Обычно к таким ресурсам выполняют запросы методом GET для просмотра списка ресурсов или методом POST для создания нового ресурса, например, новой статьи.

Также часто можно встретить ситуацию, когда для любых изменений данных, включая удаление, используется метод POST вместо PUT, PATCH и DELETE. Обычные веб-формы (html-формы), например, поддерживают только методы GET и POST для отправки данных, поэтому в случае серверного рендера страниц (server-side rendering) обычно используют только эти методы. При наличии интерфейса REST-API и отдельного клиентского приложения, сервер обычно обрабатывает все виды запросов.

Параметры GET запроса

Параметры для GET запроса передаются в адресной строке браузера после адреса сайта и пути к странице на нём. Давайте рассмотрим типичную строку из URL HTTP-запроса:

```
http://localhost:8000/articles/?author=Jack&title=Article
```

URL (Universal Resource Locator - Универсальный Локатор Ресурса) состоит из следующих компонентов:

`http://`	протокол
`localhost`	доменное имя (здесь локальное)
`:8000`	номер порта удаленного сервера

`/articles/`	путь (path)
`?author=Jack&title=Article`	строка запроса (query string)

Строка запроса содержит **параметры** запроса и состоит из пар "ключ-значение", которые разделяются между собой знаком "&" (амперсанд). Начало строки запроса обозначается знаком вопроса - "?".

Строка запроса не влияет на то, какое представление будет обрабатывать запросы по указанному адресу. При выборе представления в Django учитывается только **путь** (**path**).

Попробуем посмотреть, как параметры приходят на сервер. Чтобы получить доступ к параметрам запроса нужно использовать аргумент request, который указывается в каждом представлении.

Переменная request является объектом типа `HttpRequest` из модуля `django.http.request`. У неё есть свойства:

- GET - объект на базе словаря с данными, приходящими с строке запроса. В основном применяется для GET-запроса, но и в POST-запросах этот словарь тоже заполняется.
- POST - объект на базе словаря с данными, приходящими в теле запроса, при отправке форм методом POST.
- FILES - список файлов, загруженных пользователем через форму.

Для примера работы со словарём GET добавьте в index_view строчку

```
print(request.GET)
```

и откройте главную страницу с любыми параметрами в конце ссылки, например:

```
http://localhost:8000/?my_param=1
```

В консоли сервера вы увидите, как параметры пришли на сервер:

```
{'my_param': ['1']}
```

Значение параметра является списком, т.к. он может встречаться в строке запроса несколько раз, и Django представляет его в виде списка:

```
http://localhost:8000/?my_param=1&my_param=2
```

```
{'my_param': ['1', '2']}
```

Все параметры приходят, как строки и вам нужно вручную приводить их к нужному вам типу данных.

Чтобы достать из словаря GET значение одного параметра, используйте его метод get(), аналогичный методу get() у обычных словарей - он возвращает значение указанного ключа, либо значение по умолчанию, которое передаётся вторым аргументом:

```
http://localhost:8000/?my_param=1
```

```
request.GET.get('my_param')          # вернёт 1
request.GET.get('my_param', 10)      # все равно вернёт 1, т.к.
ключ есть
request.GET.get('other_param')       # вернёт None, т.к. такого
ключа нет
request.GET.get('other_param', 10)   # вернёт 10 - значение по
умолчанию
```

Обращение по ключу через оператор `[]` (квадратные скобки) в этих словарях не работает. Это сделано специально, чтобы вы использовали метод `get()`, и ваш код не вылетал бы с ошибкой, если какого-то параметра нет или он не передан.

Для нескольких значений метод `get()` всегда будет возвращать последнее из них

```
http://localhost:8000/?my_param=1&my_param=2

request.GET.get('my_param')          # вернёт 2
```

Поэтому для нескольких значений (там, где вы ожидаете, что может прийти список), используйте метод getlist():

```
request.GET.get_list('my_param')     # вернёт ['1', '2']
```

Параметры POST запроса

Параметры POST запросов передаются, например, при заполнении и отправке веб-форм.

Добавим ещё одно представление в **webapp/views.py**:

```
def article_create_view(request):
    if request.method == 'GET':
        return render(request, 'article_create.html')
    elif request.method == 'POST':
        print(request.POST)
```

Это представление проверяет свойство method у объекта запроса (`request.method`), в котором записан HTTP-метод, и в зависимости от него либо показывает форму добавления новой статьи, либо выводит параметры POST-запроса.

Подключим представление в urls.py на адрес `/articles/add`. Для этого импортируем его из `webapp/views.py`:

```
from webapp.views import index_view, article_create_view
```

И добавим ещё один путь в urlpatterns (не забудьте запятую после существующих путей!):

```
path('articles/add/', article_create_view)
```

Полный код `urls.py`:

```
from django.contrib import admin
from django.urls import path
```

```
from webapp.views import index_view, article_create_view

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', index_view),
    path('articles/add/', article_create_view)
]
```

Наконец, добавим шаблон `article_create.html`, который указан в представлении `article_create_view` в папку `templates`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Create</title>
</head>
<body>

<h1>Create Article</h1>

<form action="/articles/add/" method="post">
    {% csrf_token %}

    <label>Title:</label><br/>
    <input type="text" name="title"/><br/>

    <label>Text:</label><br/>
    <textarea name="content"></textarea><br/>

    <label>Author:</label><br/>
    <input type="text" name="author"/><br/>
    <br/>
    <input type="submit" value="Send"/>
</form>

</body>
</html>
```

Шаблон содержит форму добавления новой статьи: заголовок, текст, автор.

`{% csrf_token %}` - это специальный тег Django, который добавляет в форму поле со случайно сгенерированным значением, уникальным для каждого нового запроса, которое служит для защиты от т.н. CSRF-инъекций - одного из видов атак.

Если сейчас открыть страницу `http://localhost:8000/articles/add/`, заполнить форму и отправить её, вы увидите в консоли пришедшие параметры запроса:

```
{'csrfmiddlewaretoken': ['...'], 'title': ['Title'], 'content':
['Text''], 'author': ['Author']}
```

Использование переменных в шаблонах

Пока что пришедшие параметры нигде не используются, а только выводятся в консоль. Давайте заменим код в ветке 'POST' представления `article_create_view` на следующий:

```
context = {
    'title': request.POST.get('title'),
    'content': request.POST.get('content'),
    'author': request.POST.get('author')
}
return render(request, 'article_view.html', context)
```

Полный код представления `article_create_view`:

```
def article_create_view(request):
    if request.method == 'GET':
        return render(request, 'article_create.html')
    elif request.method == 'POST':
        context = {
            'title': request.POST.get('title'),
            'content': request.POST.get('content'),
            'author': request.POST.get('author')
        }
        return render(request, 'article_view.html', context)
```

Теперь представление `article_create_view` попытается срендерить шаблон `article_view.html`, передав ему определённый *контекст*.

В Django набор переменных, передаваемых в шаблон называется **контекст**. Он представляет из себя словарь значений, в котором ключи соответствуют названиям переменных, доступных в шаблоне.

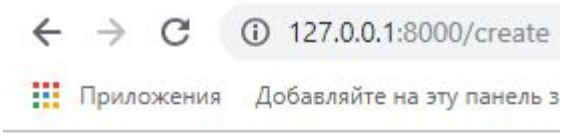
Переменные из данных запроса `request.POST` нужно доставать с помощью того же метода `get()`, что и из словаря `request.GET`.

Теперь добавим шаблон `article_view.html` в папку `templates`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>View</title>
</head>
<body>
    <h1>Future article:</h1>
    <h2>{{ title }}</h2>
    <p>{{ content }}</p>
    <p>By: {{ author }}</p>
</body>
</html>
```

Обратите внимание на специальные теги в двойных фигурных скобках: `{{ title }}`, `{{ content }}`, `{{ author }}`. Двойные фигурные скобки обозначают использование переменных из переданного в шаблон контекста.

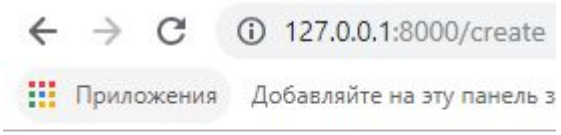
Если сейчас открыть страницу create и заново заполнить и отправить форму, то вы увидите страницу с будущей статьёй:



Create resource

Title:
Title
Text:
Text
Author:
Author

Send



Future article:

Title

Text

By: Author

Статические файлы

Статические файлы - это файлы, которые постоянно хранятся на сервере, и не меняются в зависимости от запросов. Обычно это картинки, файлы стилей и скрипты на языке javascript. К статическим файлам относятся и файлы Bootstrap и других CSS-фреймворков, если вы не подключаете их через CDN (с облачного сервера).

HTML-шаблоны в Django к статическим файлам не относятся, т.к. страницы генерируются из них с разными данными в зависимости от запросов. Даже если шаблон не содержит переменных и меняющихся данных, обрабатывается он так же,

как и другие шаблоны - через функцию render или аналогичную в представлении, т.е. не отдаётся статическим образом, сам по себе, как файл.

Для работы со статическими файлами в Django требуется указать путь, по которому эти файлы будут доступны на сайте в `settings.py`. Кроме того, требуется указать папку, где Django будет их хранить. Откройте файл `settings.py` и найдите в нём раздел:

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/2.1/howto/static-files/
```

Проверьте, что там есть следующая строчка, если нет - добавьте её:

```
STATIC_URL = '/static/'
```

`STATIC_URL` - это константа, которая определяет путь к статическим файлам на сайте.

Кроме `STATIC_URL` у django есть ещё две настройки для статических файлов - `STATICFILES_DIRS` и `STATIC_ROOT`:

- `STATICFILES_DIRS` - это список папок, где лежат статические файлы, кроме папок по умолчанию (см. далее). По умолчанию этот список пуст, но вы можете там указать дополнительные папки со статическими файлами, кроме папок в приложениях. Сейчас эта настройкам вам не нужна.
- `STATIC_ROOT` - это специальная папка, куда Django "собирает" все статические файлы со всего приложения при выполнении команды `./manage.py collectstatic`. Это необходимо на производственном сервере, чтобы не настраивать его отдельно для выдачи статиков из каждой отдельной папки в приложениях или `STATICFILES_DIRS`. `STATIC_ROOT` не должен быть указан в `STATICFILES_DIRS`. Для разработки эта настройка не требуется.

По умолчанию сервер Django собирает статические файлы из папок **static** в приложениях, и отдаёт их по адресу, указанному в STATIC_URL. Например, если у вас есть файл `app/static/css/style.css`, где `app` - это одно из ваших приложений, и `STATIC_URL = '/static/'`, то style.css на сервере разработки будет доступен по адресу http://localhost:8000/static/css/style.css.

Создайте папку `static` в приложении webapp. Теперь вы можете создавать в ней css, js, картинки и другие статические файлы. Далее создайте в папке `webapp/static` папку `css` и файл `css/style.css` и добавьте в него какие-нибудь стили, например:

```
body {
    font-family: Arial, sans-serif;
    background: rgb(222, 240, 204);
    color: #222222;
}
```

Для подключения статических файлов в шаблоны нужно использовать специальный модуль Django `static`. Для подключения дополнительных модулей в шаблонах используется тег `{% load %}` с указанием имён одного или нескольких модулей. Обычно он пишется в начале кода шаблона, но всегда после тега `{% extends %}`, если он есть. Каждый модуль может предоставлять дополнительные теги шаблонов Django и фильтры, о которых мы поговорим позже.

Подключите модуль `static` в начало вашего шаблона (шаблонов):

```
{% load static %}
```

Тег `{% load %}` нужно прописывать в каждом шаблоне, куда вы хотите подключить какой-либо модуль.

Теперь в нём будет доступен тег `{% static %}`, который выводит пути к статическим файлам.

Например, файл `css/style.css` можно подключить следующим образом:

```
<head>
    ...
    <link rel="stylesheet" href="{% static "css/style.css" %}">
</head>
```

Удалите тег `<style>`, если он есть, со всем содержимым, он больше не нужен.

Таким же образом можно подключать картинки и скрипты, в этом случае тег `{% static %}` прописывается в их атрибутах `src`, например:

```
<img src="{% static "img/my_image.png" %}" alt="my image">
<script type="text/javascript" src="{% static "js/my_script.js"
%}"></script>
```

При подключении статических файлов из интернета, например, с CDN, или картинок из сети, тег `{% static %}` не нужен. Он используется только для того, чтобы сгенерировать url из путей к локальным файлам. Например, подключение Bootstrap из Bootstrap CDN (https://www.bootstrapcdn.com/):

```
<link
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css" rel="stylesheet">

<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.j
s" crossorigin="anonymous"></script>
```

Дополнительно

- https://tools.ietf.org/html/rfc2616 - спецификация HTTP,
- https://tools.ietf.org/html/rfc7230 - более новая спецификация HTTP,
- http://lib.ru/WEBMASTER/rfc2068/ - устаревшая спецификация HTTP на русском,
- https://docs.djangoproject.com/en/2.1/ref/templates/builtins/ - теги шаблонов Django,
- https://docs.djangoproject.com/en/2.1/howto/static-files/ - документация по статическим файлам.