

Урок 2. Учим фреймворк обрабатывать GET- и POST-запросы

ОГЛАВЛЕНИЕ

[КАК разделить GET- и POST-запросы 3](#_Toc75777160)

[обработка GET-запроса 5](#_Toc75777161)

[обработка POST-запроса 6](#_Toc75777162)

[Заключение 10](#_Toc75777163)

[КЛЮЧЕВЫЕ ВЫВОДЫ 10](#_Toc75777164)



# КАК разделить GET- и POST-запросы

Пришло время научить наш фреймворк обрабатывать GET- и POST-запросы. Что это такое?

GET-запрос применяется для получения данных, например мы открываем страницу с формой ввода параметров (добавляем товар) или открываем страницу редактирования параметров уже добавленного товара.

POST-запрос применяется для отправки данных на сервер, например, мы сохраняем введенные через форму данные (по товару) в базу данных.

Но важно, что и GET-запрос позволяет отправлять данные на сервер (через параметры адресной строки).

Итак, нам нужно расширить исходный код фреймворка. Но как фреймворк поймет, где GET- и где POST-запрос? Здесь нам поможет словарь, отправляемый сервером, тот самый **environ** (см. материалы урока 1).

**Листинг 1. step\_2/simba\_framework/main.py**

|  |
| --- |
| **import** quopri **from** framework\_requests **import** GetRequests, PostRequests   **class** PageNotFound404:  **def** \_\_call\_\_(self, request):  **return '404 WHAT'**, **'404 PAGE Not Found'   class** Framework:   *"""Класс Framework - основа WSGI-фреймворка"""* **def** \_\_init\_\_(self, routes\_obj):  self.routes\_lst = routes\_obj   **def** \_\_call\_\_(self, environ, start\_response):  *# Получаем адрес, по которому пользователь выполнил переход* path = environ[**'PATH\_INFO'**]   *# Добавляем закрывающий слеш* **if not** path.endswith(**'/'**):  path = **f'{**path**}/'** request = {}  *# Получаем все данные запроса* method = environ[**'REQUEST\_METHOD'**]  request[**'method'**] = method   **if** method == **'POST'**:  data = PostRequests().get\_request\_params(environ)  request[**'data'**] = data  print(**f'Нам пришёл post-запрос:**  **{**Framework.decode\_value(data)**}'**)  **if** method == **'GET'**:  request\_params = GetRequests().get\_request\_params(environ)  request[**'request\_params'**] = request\_params  print(**f'Нам пришли GET-параметры: {**request\_params**}'**)   *# Находим нужный контроллер* **if** path **in** self.routes\_lst:  view = self.routes\_lst[path]  **else**:  view = PageNotFound404()   *# Запускаем контроллер* code, body = view(request)  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  **return** [body.encode(**'utf-8'**)]   @staticmethod  **def** decode\_value(data):  new\_data = {}  **for** k, v **in** data.items():  val = bytes(v.replace(**'%'**, **'='**).replace(**"+"**, **" "**), **'UTF-8'**)  val\_decode\_str = quopri.decodestring(val).decode(**'UTF-8'**)  new\_data[k] = val\_decode\_str  **return** new\_data |

В приведенном листинге мы получаем тип запроса, обратившись к параметру **'request\_method'** словаря **environ**. В зависимости от полученного значения мы выполняем обработку запроса.

# обработка GET-запроса

Ниже приведен листинг, в котором реализован класс-обработчик GET-запроса.

**Листинг 2. step\_2/simba\_framework/framework\_requests.py**

|  |
| --- |
| *# Обработка GET-запроса с параметрами* **class** GetRequests:   @staticmethod  **def** parse\_input\_data(data: str):  result = {}  **if** data:  *# Делим параметры через &* params = data.split(**'&'**)  **for** item **in** params:  *# Делим ключ и значение через =* k, v = item.split(**'='**)  result[k] = v  **return** result   @staticmethod  **def** get\_request\_params(environ):  *# Получаем параметры запроса* query\_string = environ[**'QUERY\_STRING'**]  *# Превращаем параметры в словарь* request\_params = GetRequests.parse\_input\_data(query\_string)  **return** request\_params |

Вспомним, что через GET-запрос также можно передавать данные. Но как извлечь эти данные из запроса? И вновь здесь поможет словарь **environ**. В нем находится параметр **'query\_string'**, через который можно «достучаться» до строки с параметрами GET-запроса. Например, если мы хотим получить параметры, переданные через такой GET-запрос.

<http://127.0.0.1:8080?id=1&category=10>

То получим строку «id=1&category=10». Эти параметры мы можем «распарсить», т.е. извлечь из строки, поместить в словарь, а затем, например, передать обработчику запроса (контроллеру).

Контроллер может использовать эти параметры для выполнения запросов к базам данных – например, для добавления записи в БД или выполнения выборки из БД.

Код функции **parse\_input\_data()** очень простой. Функция разбирает параметры и строки и формирует из нее словарь.

Этот словарь наш фреймворк «поместит» в другой словарь – **requests**.

request[**'data'**] = data

Этот словарь **requests** мы передадим в соответствующий контроллер.

code, body = view(request)

Кроме того, наш фреймворк просигнализирует, что получен GET-запрос с параметрами.

print(**f'Нам пришёл post-запрос: {**Framework.decode\_value(data)**}'**)

Т.к. словарь **requests** мы будем передавать контроллеру, нужно настроить принятие этого словаря самим контроллером.

**Листинг 3. step\_2/views.py**

|  |
| --- |
| *"""Модуль, содержащий контроллеры веб-приложения"""* **from** simba\_framework.templator **import** render   **class** Index:  **def** \_\_call\_\_(self, request):  **return '200 OK'**, render(**'index.html'**)   **class** About:  **def** \_\_call\_\_(self, request):  **return '200 OK'**, **'about'** |

# обработка POST-запроса

Что касается обработки POST-запроса, то здесь ситуация похожа. Но есть важное отличие, что когда мы отправляем какие-то параметры через форму и срабатывает POST-запрос, то переданные через этот запрос данные идут в байтах, а значит просто так получить введенные параметры мы не сможем. Сначала байты нужно декодировать.

**Листинг 4. step\_2/simba\_framework/framework\_requests.py**

|  |
| --- |
| *# Обработка POST-запроса с параметрами* **class** PostRequests:   @staticmethod  **def** parse\_input\_data(data: str):  result = {}  **if** data:  *# Делим параметры через &* params = data.split(**'&'**)  **for** item **in** params:  *# Делим ключ и значение через =* k, v = item.split(**'='**)  result[k] = v  **return** result   @staticmethod  **def** get\_wsgi\_input\_data(env) -> bytes:  *# Получаем длину тела* content\_length\_data = env.get(**'CONTENT\_LENGTH'**)  *# Приводим к int* content\_length = int(content\_length\_data) **if** content\_length\_data **else** 0  *# Считываем данные, если они есть* data = env[**'wsgi.input'**].read(content\_length) **if** content\_length > 0 **else b''  return** data   **def** parse\_wsgi\_input\_data(self, data: bytes) -> dict:  result = {}  **if** data:  *# Декодируем данные* data\_str = data.decode(encoding=**'utf-8'**)  *# Собираем их в словарь* result = self.parse\_input\_data(data\_str)  **return** result   **def** get\_request\_params(self, environ):  *# Получаем данные* data = self.get\_wsgi\_input\_data(environ)  *# Превращаем данные в словарь* data = self.parse\_wsgi\_input\_data(data)  **return** data |

Но для начала, как вообще получить тело запроса в байтах? Мы видим следующую функцию:

**def** get\_wsgi\_input\_data(env) -> bytes:  
 *# получаем длину тела* content\_length\_data = env.get(**'CONTENT\_LENGTH'**)  
 *# приводим к int* content\_length = int(content\_length\_data) **if** content\_length\_data **else** 0  
 *# считываем данные, если они есть* data = env[**'wsgi.input'**].read(content\_length) **if** content\_length > 0 **else b''  
 return** data

И ее вызов:

data = self.get\_wsgi\_input\_data(environ)

Вновь фигурирует словарь **environ**. С помощью ключа **'content\_length'** мы получаем длину размера тела, прикрепленного к запросу. Если тело запроса существует, то приводим длину к целочисленному формату, а далее, опираясь опять же на словарь **environ**, считываем данные.

По сути ключ **'wsgi.input'** связывает нас с телом HTTP-запроса. Это тело запроса мы считываем и возвращаем в байтах. Далее полученные байты декодируются в строку и трансформируются в словарь с помощью уже знакомой по GET-запросу функции **parse\_input\_data()**. Но в файле **main.py** нашего фреймворка есть еще одна интересная строка.

print(**f'Нам пришёл post-запрос: {**Framework.decode\_value(data)**}'**)

Кто мешает нам просто вывести словарь с данными POST-запроса привычным образом.

print(**f'Нам пришёл post-запрос: {**data**}'**)

Если мы так сделаем и запустим наш проект, выполнив отправку данных POST-запросом, то получим что-то вроде.

Нам пришёл post-запрос: {'name': '%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C', 'email': 'user%40yandex.ru', 'location': 'ny'}

Т.е. словарь строится, но для некоторых символов, например, на Кириллице, представление не соответствует исходному состоянию. Нам нужно получить значения из словаря и для каждого выполнить замену символов '%', '=' на "+", " "и превратить получившуюся строку в байты функцией **bytes()**.

Если мы попытаемся сразу раскодировать данные командой:

val\_decode\_str = val.decode(**'UTF-8'**)

То получим что-то вроде:

Нам пришёл post-запрос: {'name': '=D0=B2=D0=BA=D1=80=D1=83=D0=BA=D1=83', 'email': 'cdi9=40yandex.ru', 'location': 'ny'}

Эти непонятные символы относятся к кодировке "quoted-printable". Это система кодирования двоичного текста в текст, использующая печатаемые ASCII-символы и применяемая для кодирования данных в почтовых сообщениях.

Поэтому мы применяем библиотеку **quopri**, а точнее ее функцию **decodestring()**, которая обеспечит прием байтов и перевод их же в байты, которые мы сможем далее с помощью функции **decode()** привести к привычному строковому виду в UTF-формате.

Почему вообще возникают подобные «кракозябры»?

Потому что браузеры выполняют преобразование строки в удобный вид с русскими буквами. При этом любые буквы, отличающиеся от английских, трансформируются через юникод.

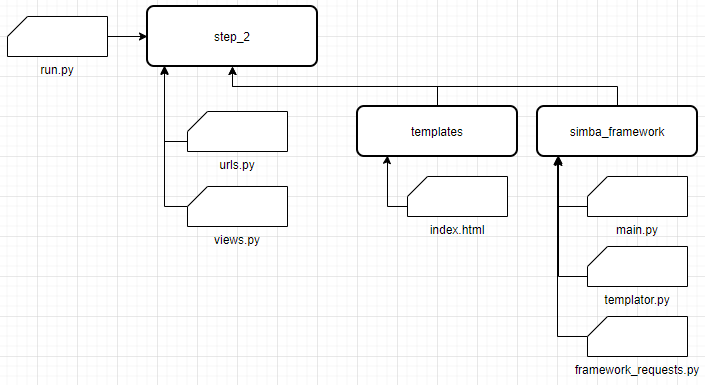
# Вид сверху на рукопожатие над деловыми документамиЗаключение

В рамках второго урока мы научили наш фреймворк разделять и обрабатывать GET- и POST-запросы. Теперь мы сможем открывать формы для определения характеристик различных сущностей и редактирования этих характеристик, а также выполнять отправку данных, введенных через формы, например, в файл, базу данных.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Научить наш фреймвор понимать где GET-, а где POST-запрос несложно. Тип запрос скрывается в уже известном нам словаре **environ**.
* Основная задача GET-запроса – получение данных, например, когда мы открываем форму редактирования уже существующей записи.
* Через POST-запрос мы выполняем отправку данных, указанных через форму.
* Теперь мы сможем реализовать в нашем веб-приложении возможность создания новых записей и редактирования существующих, например, сможем создавать новые категории курсов, сами курсы и т.д.

Текущая структура предложенного WSGI-фреймворка и создаваемого на его основе веб-приложения приведена на рис. 1.

Рис. 1. Структура проекта (фреймворк + веб-приложение)