

Урок 1. Создаем основу фреймворка и начальный функционал веб-приложения

ОГЛАВЛЕНИЕ

[КАК работают wsgi-фреймворки 3](#_Toc75526516)

[КАК создать свой wsgi-фреймворк 4](#_Toc75526517)

[КАК реализовать связку маршрутов и обработчиков 7](#_Toc75526518)

[КАК настроить простейшую шаблонизацию 8](#_Toc75526519)

[КАК реализовать запуск проекта 12](#_Toc75526520)

[Заключение 14](#_Toc75526521)

[КЛЮЧЕВЫЕ ВЫВОДЫ 14](#_Toc75526522)



# КАК работают wsgi-фреймворки

Пользователь через интерфейс веб-приложения переходит на какую-либо страницу. При этом формируется запрос, который отправляется на сторону веб-сервера (Apache, Nginx и т.д.).

Веб-сервер принимает запрос, который необходимо обработать. Т.к. каждый веб-сервер имеет свои особенности, то пришлось бы создавать веб-приложение отдельно под каждый веб-сервер. Необходим протокол, по которому эти сервера (балансировщики нагрузки) универсально выполняли бы обработку запросов. Таким протоколом является WSGI.

Параметры запроса, отправленного пользователем, необходимо доставить до Python-приложения. Но Python-приложению технически проще будет работать не с данными (байтами), а, например, со словарем. Веб-сервер по протоколу WSGI передает байты WSGI-коннектору (библиотеке Gunicorn или uWSGI).

WSGI-коннектор конвертирует байты в словарь и передает Python-приложению, которое выполняет с ним необходимые действия. Например, на основе содержимого словаря выполняет запрос к БД. Важно отметить, что под Python-приложением мы будем понимать и код фреймворка, и код веб-приложения (а точнее код той его части, которая называется бэкендом и написана на Python).

Приложение на Python оформляет ответ, который передается на сторону WSGI-коннектора (Gunicorn или uWSGI), далее ответ отправляется веб-серверу и в конечном счете ответ отображается в браузере.

На рис. 1 приведена связка компонентов, обеспечивающих работу веб-приложения.

Рис. 1. Связка компонентов, обеспечивающих работу веб-приложения

# КАК создать свой wsgi-фреймворк

Первая задача, которая стоит перед нами, заключается в разработке фреймворка, т.е. WSGI-приложения. Работу таких приложений регламентирует стандарт PEP-3333. В соответствии с данным стандартом WSGI-приложение обладает следующими характеристиками.

1. Это вызываемый объект (callable-объект). Вызываемым объектом в Python называется функция или объект класса, у которого реализована перегрузка метода **\_\_call\_\_()**.
2. Принимает в качестве аргументов словарь (**environ**) с параметрами запроса и обработчик запроса (**start\_response()**). Словарь поступает от WSGI-коннектора, а обработчик запроса позволяет выполнить ответ на поступивший запрос.
3. Обеспечивает вызов обработчика запроса. Обработчик принимает код HTTP-ответа и HTTP-заголовок.
4. Обеспечивает возврат итерируемого объекта с телом ответа.

Дадим нашему фреймворку название **simba\_framework** и создадим для него отдельную директорию. В этой директории мы будем размещать модули исходного кода фреймворка.

**Листинг 1. step\_1/simba\_framework/main.py**

|  |
| --- |
| **class** PageNotFound404:  **def** \_\_call\_\_(self):  **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**}/'** *# Находим нужный контроллер* **if** path **in** self.routes\_lst:  view = self.routes\_lst[path]  **else**:  view = PageNotFound404()   *# Запускаем контроллер* code, body = view()  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  **return** [body.encode(**'utf-8'**)] |

WSGI-приложение мы реализуем в виде объекта класса с перегрузкой метода **\_\_call\_\_()**. В приведенном классе присутствует и перегрузка метода **\_\_init\_\_()** с передачей интересного аргумента **routes\_obj**. К тому, для чего он нужен, мы вернемся позднее.

Итак, вспомним, что наш callable-объект принимает словарь **environ**, в котором мы можем найти много важных параметров запроса. Например, мы можем получить путь, указанный пользователем в адресной строке браузера и запустить соответствующую функцию-обработчик. Т.е. мы можем научить наш фреймворк запускать контроллер, соответствующий каждому URL-у.

Теперь вернемся к переменной **routes\_obj**. Она ссылается на словарь, например,

routes = {  
 **'/'**: Index(),  
 **'/about/'**: About(),  
}

Этот словарь уже относится к нашему будущему веб-приложению и будет находиться в исходном коде веб-приложения, а не фреймворка. Если вы изучали Django и Flask, то заметите сходство с маршрутизацией, реализуемой в этих фреймворках. Только в Django это называется «привязками», т.е. к каждому пути привязана своя функция-контроллер, а во Flask это роуты (маршруты).

Т.е. наш фреймворк обеспечит запуск соответствующего каждому пути контроллера.

Как же он это сделает? Все просто. Мы получаем путь, который указал пользователь в адресной строке:

path = environ[**'PATH\_INFO'**]

А дальше в словаре ищем соответствующий ему объект класса-контроллера:

**if** path **in** self.routes\_lst:

view = self.routes\_lst[path]

**else**:

view = PageNotFound404()

И вызываем этот объект класса как функцию:

code, body = view(request)

Обработчик запроса **start\_response()** отправит код ответа, а сам фреймворк передаст тело ответа в байтах через WSGI-коннектор на сторону веб-сервера, который в свою очередь передаст его клиенту.

Если же нужного контроллера в словаре не окажется, будет создан и вызван объект специального служебного класса, обеспечивающего вывод информации, что страница не найдена (код 404).

# КАК реализовать связку маршрутов и обработчиков

На текущем уроке мы дополним фреймворк еще одним модулем. Но об этом позднее. А теперь по аналогии с Django-проектами создадим модули **urls.py** и **views.py**. Содержимое файлов приведено ниже.

**Листинг 2. step\_1/views.py**

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

**Листинг 3. step\_1/urls.py**

|  |
| --- |
| **from** views **import** Index, About  *# Набор привязок: путь-контроллер* routes = {  **'/'**: Index(),  **'/about/'**: About(), } |

Содержимое файла **urls.py** мы уже разобрали. Теперь рассмотрим, что находится в файле **views.py**. Здесь находится код классов-контроллеров, обеспечивающих обработку запросов. Контроллер **About()** отправляет пользователю простой ответ, а контроллер **Index()** обеспечивает рендеринг шаблона. Что означает эта процедура, разработчики, работавшие с Django и Flask, точно знают. Но обеспечивающая рендеринг функция **render()** встроенная или пользовательская?

Эта функция пользовательская, но судя по импорту:

**from** simba\_framework.templator **import** render

Она относится не к нашему веб-приложению, а к фреймворку. А значит самое время дополнить наш фреймворк еще одним модулем – **templator**, в котором мы научим наш фреймворк обеспечивать рендеринг шаблонов.

# КАК настроить простейшую шаблонизацию

|  |
| --- |
| **from** os.path **import** join **from** jinja2 **import** Template   **def** render(template\_name, folder=**'templates'**, \*\*kwargs):  *"""* **:param** *template\_name: имя шаблона* **:param** *folder: папка в которой ищем шаблон* **:param** *kwargs: параметры, передаваемые в шаблон* **:return***:  """* file\_path = join(folder, template\_name)  *# Открываем шаблон по имени* **with** open(file\_path, encoding=**'utf-8'**) **as** f:  *# Читаем* template = Template(f.read())  *# Рендерим шаблон с параметрами* **return** template.render(\*\*kwargs) |

**Листинг 4. step\_1/simba\_framework/templator.py**

Видим такой интересный импорт:

**from** jinja2 **import** Template

Для того, чтобы выполнять рендеринг шаблонов и передавать в них динамические данные (контекст), нам необходимо приложение-шаблонизатор. В Django шаблонизатор встроенный. Нам нужно либо создать свой шаблонизатор, либо подключить сторонний. Написать свой шаблонизатор – задача не из легких, поэтому воспользуемся сторонним.

Воспользуемся шаблонизатором Jinja2. Он включен во Flask, но может быть установлен и как самостоятельное приложение.

Установим Jinja2 командой:

pip install jinja2

Для обеспечения рендеринга шаблонов мы напишем функцию **render()**, принимающую несколько аргументов.

template\_name – имя шаблона, рендеринг которого предполагается выполнить.

folder – папка, в которой выполняется поиск шаблонов.

\*\*kwargs – параметры, которые мы можем передать в шаблон. Это так называемые динамически передаваемые данные (контекст).

Далее, используя класс **Template,** мы обеспечиваем открытие и считывание содержимого шаблона в строку.

И наконец мы вызываем уже встроенную функцию **render()**, обеспечивающую отображение шаблона.

Важно, чтобы шаблоны находились в папке **templates**. В этой папке мы пока сохраним один шаблон.

**Листинг 5. step\_1/templates/index.html**

|  |
| --- |
| <!DOCTYPE **html**> <**html**>  <**head**>  <**title**>Школа программирования</**title**>  <**style type="text/css"**>  body {  font-family: Arial, Verdana, sans-serif;  font-size: 90%;  color: #666;  background-color: #f8f8f8;}  li {  list-style-image: url("images/icon-plus.png");  line-height: 1.6em;}  table {  border-spacing: 0px;}  th, td {  padding: 5px 30px 5px 10px;  border-spacing: 0px;  font-size: 90%;  margin: 0px;}  th, td {  text-align: left;  background-color: #e0e9f0;  border-top: 1px solid #f1f8fe;  border-bottom: 1px solid #cbd2d8;  border-right: 1px solid #cbd2d8;}  tr.head th {  color: #fff;  background-color: #90b4d6;  border-bottom: 2px solid #547ca0;  border-right: 1px solid #749abe;  border-top: 1px solid #90b4d6;  text-align: center;  text-shadow: -1px -1px 1px #666666;  letter-spacing: 0.15em;}  td {  text-shadow: 1px 1px 1px #ffffff;}  tr.even td, tr.even th {  background-color: #e8eff5;}  tr.head th:first-child {  -webkit-border-top-left-radius: 5px;  -moz-border-radius-topleft: 5px;  border-top-left-radius: 5px;}  tr.head th:last-child {  -webkit-border-top-right-radius: 5px;  -moz-border-radius-topright: 5px;  border-top-right-radius: 5px;}  fieldset {  width: 310px;  margin-top: 20px;  border: 1px solid #d6d6d6;  background-color: #ffffff;  line-height: 1.6em;}  legend {  font-style: italic;  color: #666666;}  input[type="text"] {  width: 120px;  border: 1px solid #d6d6d6;  padding: 2px;  outline: none;}  input[type="text"]:focus,  input[type="text"]:hover {  background-color: #d0e2f0;  border: 1px solid #999999;}  input[type="submit"] {  border: 1px solid #006633;  background-color: #009966;  color: #ffffff;  border-radius: 5px;  padding: 5px;  margin-top: 10px;}  input[type="submit"]:hover {  border: 1px solid #006633;  background-color: #00CC33;  color: #ffffff;  cursor: pointer;}  .title {  float: left;  width: 160px;  clear: left;}  .submit {  width: 310px;  text-align: right;}  </**style**>  </**head**>  <**body**>  <**h1**>Наши курсы</**h1**>  <**p**>Все программы действуют в течение года.</**p**>  <**table**>  <**tr class="head"**>  <**th**></**th**>  <**th**>Москва</**th**>  <**th**>Санкт-Петербург</**th**>  <**th**>Екатеринбург</**th**>  </**tr**>  <**tr**>  <**th**>Основы Python</**th**>  <**td**>Сб, 20 февраля 2021<**br** />11:00 - 14:00</**td**>  <**td**>Сб, 6 марта 2021<**br** />11:00 - 14:00</**td**>  <**td**>Сб, 20 марта 2021<**br** />11:00 - 14:00</**td**>  </**tr**>  <**tr class="even"**>  <**th**>Алгоритмы и структуры данных на Python</**th**>  <**td**>Сб, 10 апреля 2021<**br** />11:00 - 13:00</**td**>  <**td**>Сб, 1 мая 2021<**br** />11:00 - 13:00</**td**>  <**td**>Сб, 8 мая 2021<**br** />11:00 - 13:00</**td**>  </**tr**>  <**tr**>  <**th**>Основы Django</**th**>  <**td**>Сб, 26 июня 2021<**br** />11:00 - 14:00</**td**>  <**td**>Сб, 3 июля 2021<**br** />11:00 - 14:00</**td**>  <**td**>Сб, 10 июля 2021<**br** />11:00 - 14:00</**td**>  </**tr**>  <**tr class="even"**>  <**th**>Основы Flask</**th**>  <**td**>Сб, 28 августа 2021<**br** />11:00 - 16:00</**td**>  <**td**>Сб, 4 сентября 2021<**br** />11:00 - 16:00</**td**>  <**td**>Сб, 11 сентября 2021<**br** />11:00 - 16:00</**td**>  </**tr**>  </**table**>  <**form action="" method="post"**>  <**fieldset**>  <**legend**>Зарегистрируйтесь</**legend**>  <**p**><**label class="title" for="name"**>Ваше имя:</**label**>  <**input type="text" name="name" id="name"**><**br** />  <**label class="title" for="email"**>Ваш email:</**label**>  <**input type="text" name="email" id="email"**></**p**>  <**p**><**label for="location" class="title"**>Ближайший центр:</**label**>  <**select name="location" id="location"**>  <**option value="ny"**>Москва</**option**>  <**option value="il"**>Санкт-Петербург</**option**>  <**option value="ca"**>Екатеринбург</**option**>  </**select**></**p**>  <**span class="title"**>Являетесь участником?</**span**>  <**label**><**input type="radio" name="member" value="yes"** /> Да</**label**>  <**label**><**input type="radio" name="member" value="no"** /> Нет</**label**></**p**>  </**fieldset**>  <**div class="submit"**><**input type="submit" value="Регистрация"** /></**div**>  </**form**>  </**body**> </**html**> |

Наш курс ориентирован в первую очередь на работу с backend-составляющей проекта, поэтому свои HTML-шаблоны мы не будем придумывать, а воспользуемся готовыми.

# КАК реализовать запуск проекта

Первый шаг работы над проектом мы практически завершили. Но не хватает еще одного важного момента – как запустить веб-приложение, написанное на основе нашего фреймворка? Хотелось бы сделать что-то похожее на запуск как в Django.

Вспомним, что для запуска нам необходим веб-сервер и WSGI-коннектор. Эту инфраструктуру обеспечит модуль **wsgiref**, благодаря которому мы сможем запустить приложение на нашем localhost.

**Листинг 5. step\_1/run.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server **from** simba\_framework.main **import** Framework **from** urls **import** routes  *# Создаем объект WSGI-приложения* application = Framework(routes)  **with** make\_server(**''**, 8080, application) **as** httpd:  print(**"Запуск на порту 8080..."**)  httpd.serve\_forever() |

В приведенном листинге мы создаем объект нашего WSGI-приложения с передачей набора URL-маршрутов. Напомним, что эти маршруты мы задали в модуле **urls**. В приведенном листинге функция **make\_server()** обеспечивает создание нового WSGI-сервера, который будет прослушивать указанный хост и порт, принимая подключения для нашего приложения (**application**) и выполняя обработку запросов.

Выполним запуск файла **run.py** из PyCharm с помощью комбинации клавиш ctrl+shift+f10. Перейдем в браузер и введем в адресной строке:



Результат:

# Вид сверху на рукопожатие над деловыми документамиЗаключение

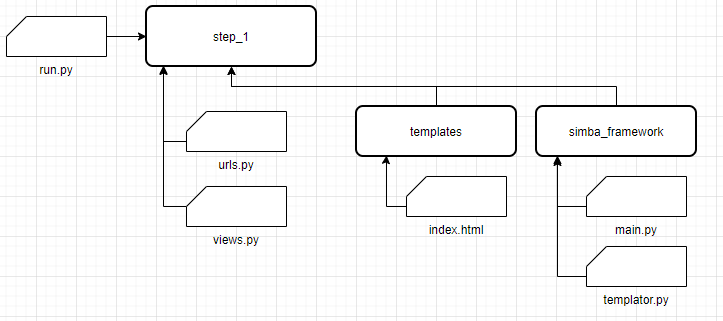
Друзья, наш первый урок подошел к концу. Теперь вы знаете как создать основу для фреймворка и научить его тем базовым возможностям, которые реализованы в Django и Flask.

Мы подготовили фундамент нашего фреймворка и даже реализовали простейший функционал веб-приложения на базе этого фреймворка. С каждым уроком мы будем расширять исходный код и возможности и фреймворка, и веб-приложения.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Фреймворк в минимальной реализации – это callable-объект, который должен обеспечивать прежде всего обработку поступающих запросов.
* Запросы и связанные с ними данные поступают от веб-сервера в байтовом формате, но за счет WSGI-коннекторов транслируются в более привычный формат Python-объектов (словарей).
* С указанным словарем фреймворк выполняет необходимые операции, например, находит извлекает путь, введенный пользователем в адресной строке и запускает соответствующий контроллер.
* Ответ в байтовом формате через WSGI-коннектор отправляется обратно на сервер, а далее транслируется пользователю.

Текущая структура предложенного WSGI-фреймворка и создаваемого на его основе веб-приложения приведена на рис. 2.

Рис. 2. Структура проекта (фреймворк + веб-приложение)