*Урок 1.* [*Паттерны web-представления*](https://gb.ru/lessons/192653)

*Введение в курс*

Наш курс ориентирован в большей степени в сторону веб-разработки, т.к. это одно из наиболее перспективных направлений в области программирования и конечно это та область, где успешно применяется Python для разработки бэкенда приложений. Цель курса заключается в освоении паттернов, т.е. некоторых шаблонов (подходов) реализации программных продуктов, а также в изучении возможностей их применения на примере своего проекта.

Речь идет прежде всего о паттернах, применяемых в веб-программировании. По итогам курса мы создадим собственный оригинальный WSGI-фреймворк – упрощенную копию фреймворков Django и Flask. Подключим сторонний шаблонизатор. Реализуем свою ORM. Курс обновленный, он изменился по сравнению с прежней версией и теперь у нас 9 уроков вместо 8. Добавился еще урок по микросервисам.

В рамках курса мы освоим основные варианты архитектур приложений, и вы обязательно сможете применить эти знания в дальнейшем и в ходе обучения (например, на командной разработке), и в реальной работе.

Разумеется, мы разберем и закрепим фундаментальные шаблоны проектирования, а также улучшим свои знания и умения в области ООП.

Таким образом, мы освоим паттерны, реализуем мини-фреймворк, сделаем с помощью него свой проект (небольшой сайт, предлагающий обучающие курсы).

В курсе будет немало абстрактных вещей и теории, но не пугайтесь, это нам обязательно пригодится в работе.

*Что даст нам курс*

Применять в работе готовые, проверенные, эффективные шаблоны, а не изобретать новые. Повысить свой статус разработчика – разговаривать с коллегами на одном языке и уметь продемонстрировать ценные умения, стать более ценным специалистом. Курс развивает наше мышление разработчика и архитектора, позволяет быстро определить, использовать ли нам некоторое готовое решение или работать над новым.

*Паттерны веб-представления*

На сегодняшнем уроке мы вспомним (или освоим, если вы еще не встречались с этим ранее), в чем смысл паттернов MVC и MVT, узнаем, что есть такие паттерны, как Page Controller и Front Controller, разберемся, как функционируют WSGI-фреймворки и даже начнем писать свой WSGI-фреймворк, обязательно используя в нем паттерны Page Controller и Front Controller.

*Паттерн MVC/MVT*

Сразу начнем с очень популярного паттерна, о котором наверняка все слышали. Он называется MVC.

Сравним классическую схему MVC и концепцию MVT, которая реализуется именно в Django.

Основная идея заключается в том, что мы разделяем приложение на три уровня:

**Model**. Включает бизнес-логику приложения. Она может быть связана с БД. Здесь же мы говорим об ORM. Бизнес-логика включает работу с данными и бизнес-процессами.

**Controller**. Обработка запросов.

**View**. Интерфейс приложения. То, что видит пользователь, но необязательно это HTML.

Контроллер знает про Model и про View. Модель ни про кого не знает, она независима. View тоже можно менять без изменения модели.

В Django эта концепция реализована по-другому. Models – это наши классы – шаблоны таблиц БД, унаследованные от model.models. Views – это «вьюшки», реализованные через функции или классы. Template – шаблоны html-страниц.

Может ли приложение работать без MVC? Да, может, классический пример – страница на PHP, напрямую делающая запрос в БД. Тут получается только два уровня. Страница и в ней дизайн, и логики, и все вместе, и она еще делает запросы в БД. Но такая схема неудобная и подходит только для одностраничников.

Поэтому шаблон MVC с расслоением приложения гораздо эффективнее и популярнее для реальных, больших проектов.

Что касается Django, то там есть так называемые «урлы» (или «роуты» во Flask).

**Достоинства MVC, в чем они?**

Модель – это ядро нашей системы, она автономна и тестируема, там наша бизнес-логика, поэтому желательно, чтобы она ни от чего не зависела. Там мы будем описывать наши данные и правила.

Контроллер – это непосредственно логика работы нашего сайта. Пришел запрос, нам нужно узнать куда его отправить и что делать.

Представления – отображают данные. Тоже хорошо, т.к. мы можем сделать, например, две разные страницы для одной модели.

**Что дает MVC?**

Множественность вариаций представлений. Мы можем для одной модели сделать несколько представлений (интерфейсов).

Концентрация усилий. Можно распределить разработчиков по различным задачам.

Параллельная разработка. Специалисты могут одновременно работать над своими различающимися задачами.

Простота модификации. Разбиение на слои, в том числе независимые, позволяет при необходимости оперативно внести нужные изменения.

MVC – фреймворки. Самой собой, это Django. Во Flask это тоже есть, но там нет моделей. Можно подключить стороннюю ORM-библиотеку SQLAlchemy и создавать модели, в качестве шаблонизатора используем Jinja2. Концепция MVC прижилась прежде всего в вебе, потому что в вебе все работает по запросу. Пользователь отправляет запрос и после этого уже что-то происходит.

*Паттерн Page Controller*

Это, по сути, то, с чем вы работали в Django.

На сайте есть адреса. Пользователи отправляют на них запросы. К адресам привязаны «вьюшки» (функции или классы).

**Плюсы:**

Простота. Мы берем «урлы», «вьюшку» и она что-то делает, очень удобно. Нам нужен новый функционал? Делаем «урл», «вьюшку» и готово.

Прямолинейность. Есть «урл», «вьюшка», смотрим, какая в ней логика.

Отделяет логику от представления. Мы делаем рЕндеринг шаблона и отдаем в него данные (контекст).

## *Паттерн Front Controller*

Про этот паттерн часто забывают. Это контроллер, обрабатывающий все запросы к приложению. Дает возможность реализовать некоторые общие правила в проекте. Эта «вьюшка» знает про все запросы.

**Данный паттерн дает нам следующие преимущества:**

Контроль. Мы можем контролировать, обрабатывать сразу все запросы.

Координация. Выполнять какие-то действия между запросами.

Конфигурируемость. Выполнять какие-то свои настройки.

Важная особенность! В Джанго используются сразу два паттерна.

Паттерн FC реализован в виде слоев Middlewares. Это слои FC. У нас слой работает в одну сторону и в другую. Пример – csrf – токен. В одну сторону добавляется токен, в другую – он проверяется. Для обработки запросов мы можем добавлять свои middlerwares – свои слои. Причем, слои можно добавлять, как в середину, так и на самый верх. Добавление слоев внутрь фронт-контроллера. И через эти слои будут проходить все запросы. Например, в Джанго, Django-debug-toolbar.

## *Как работает WSGI-фреймворк*

Приступаем к самому главному! Заглянем в устройство фреймворков (узнаем, что «под капотом») и начнем писать свой фреймворк.

У нас есть сервер, например, Nginx или Apache. И есть такая штука, под названием WSGI. По сути, это некоторая договоренность (протокол), по которому эти веб-сервера (балансировщики нагрузки), обрабатывающие запросы.

К серверам приходят запросы и по этому протоколу сервера отдают данные для приложений.

Для чего нужен WSGI? Когда его не было, приходилось писать приложения отдельно под Apache, Nginx, и т.д. И решили, почему бы не сделать протокол, по которому сервера будут отдавать данные.

Т.е. пользователь «стучится», сервер передает данные, которые кто-то может читать. Обычно, эти данные – просто набор байтов.

У нас есть приложение на Python, но данные приходят в байтах. Нужен обработчик, который превратит байты в нормальный, читабельный вид, например, в тип **dict**.

Этим и занимается Gunicorn. Или другая популярная библиотека uWSGI.

**ИТОГО:** От пользователя поступает запрос, Nginx передает данные (байты), которые через протокол WSGI принимаются WSGI-коннектором, например, Gunicorn (или uWSGI) и трансформируются в словарь, а словарь передается приложению на Python (это будет наш фреймворк).

И в обратную сторону. Приложение на Python должно отдать ответ в Gunicorn, который перенаправит его в Nginx.

*Итак, на веб-сервер поступает запрос. Сервер в определенном виде отдает данные в формате WSGI. Отдает WSGI-коннектору Gunicorn, который преобразует их в словарь или любой другой объект. Главное, чтобы его было возможным прочесть. На стороне Python-приложения мы читаем этот словарь. Что-то делаем и отдаем обратно. Супервайзер поднимает службу (Gunicorn), если сервер падает.*

*Создаем шаблон WSGI-фреймворка*

В следующем листинге представлена заготовка (шаблон) нашего фреймворка.

В соответствие с PEP-3333 (<https://www.python.org/dev/peps/pep-3333/>) нам декларируется, что должен быть некоторый callable-объект. Это функция или объект класса с перегрузкой метода \_\_call\_\_.

**Листинг 1. Урок 1. Коды к уроку/simple\_wsgi.py**

|  |
| --- |
| **def** application(environ, start\_response):  *"""* **:param** *environ: словарь данных от сервера* **:param** *start\_response: функция для ответа серверу  """  # сначала в функцию start\_response передаем код ответа и заголовки* start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  *# возвращаем тело ответа в виде списка из bite* **return** [**b'Hello world from a simple WSGI application!'**]  *# Для запуска можно использовать gunicorn или uwsgi или их аналоги  # gunicorn - wsgi-коннектор # pip install gunicorn # gunicorn simple\_wsgi:application  # uwsgi # pip install uwsgi # uwsgi --http :8000 --wsgi-file simple\_wsgi.py* |

Первый параметр – **environ**. Это словарь, который нам предоставил **Gunicorn**. Второй параметр – функция **start\_response**. Ее тоже даст **Gunicorn**. Она нужна для того, чтобы в нее мы передали код ответа и заголовок.

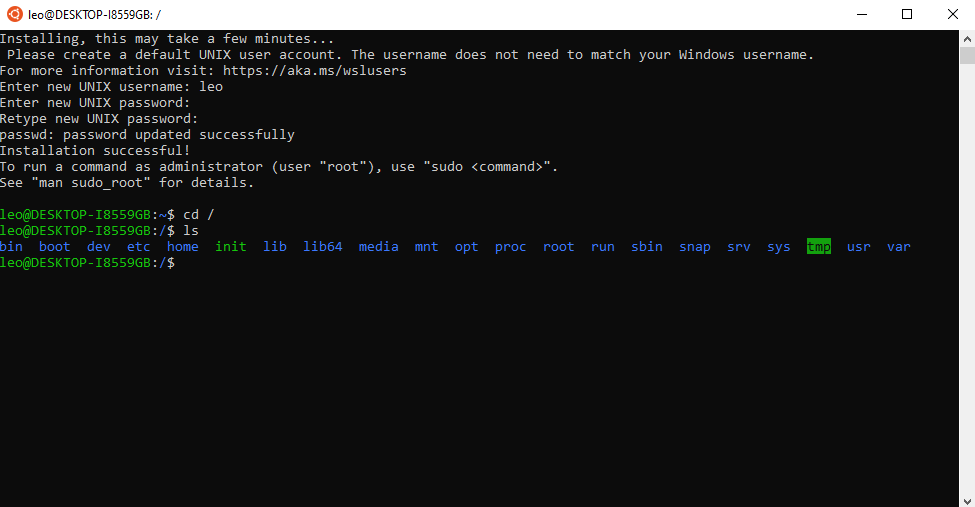
Теперь мы можем соединить этот callable-объект с **Gunicorn** или **Uwsgi**.

Мы запускаем **Gunicorn**, указываем приложение (def application()), которое будет работать с **Gunicorn**.

Начинаем запуск! Нам нужен WSGI-коннектор, т.е. либо **Gunicorn**, либо **Uwsgi**. Проблема в том, что они не работают под Windows. Можно поставить виртуалку Linux, но она потребляет много ресурсов.

Есть решение для Windows 10!

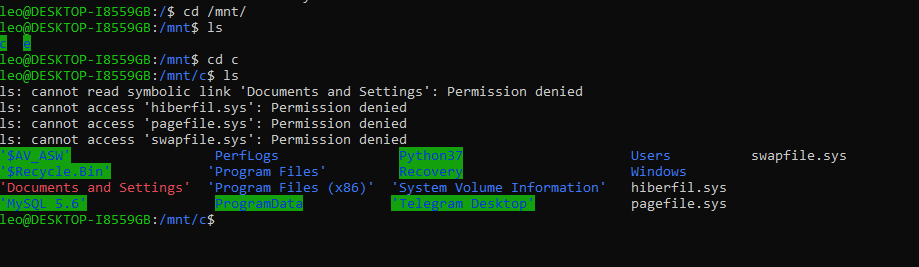
Пуск -> Microsoft Store -> Поиск -> Вводим ubuntu -> Получаем урезанную виртуальную машину, которая работает с Windows 10.

**

Кстати при возникновении проблем с установкой воспользуйтесь инструкцией - <http://windowsbulletin.com/ru/fix-wslregisterdistribution-failed-with-error-0x8007019e-and-0x8000000d/>.

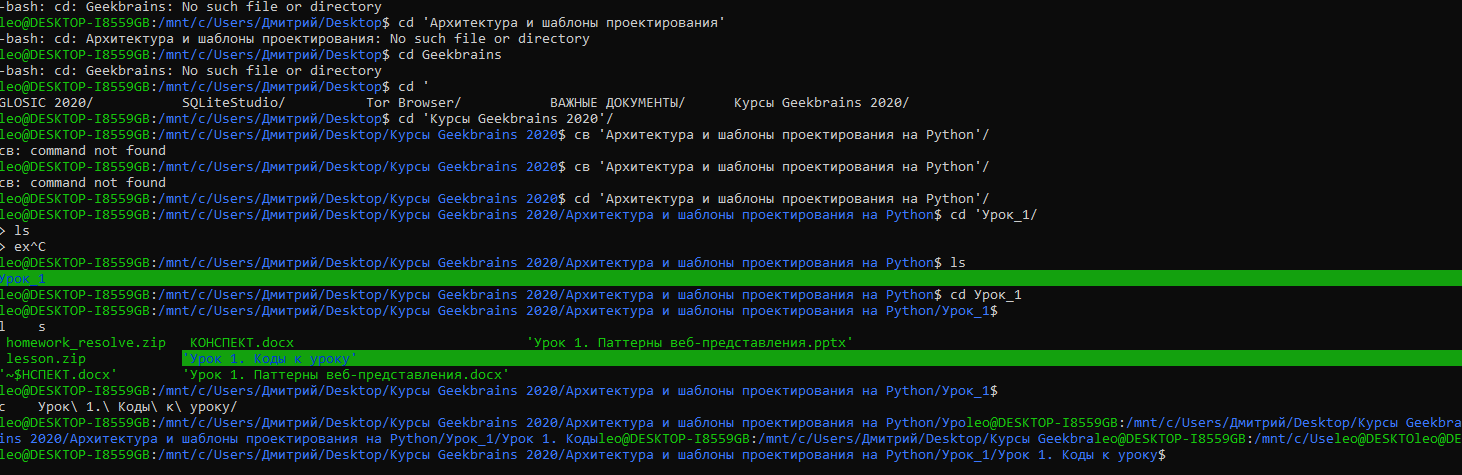
П.С. – не забудьте включить виртуализацию в BIOS.

Здесь мы можем перемещаться как будто перемещаемся по файловой системе Windows.

**

Не нужно «шарить» папки, можно обращаться напрямую.

Приступаем к запуску!



*Запускаем приложение*

Запускать будем скрипт в модуле **fwsgi.py**.

Для начала запускать будем через Uwsgi. Но эту библиотеку сперва нужно установить. А это танцы с бубном. Будут нужны некоторые доп. библиотеки.

sudo add-apt-repository universe

sudo apt update

sudo apt install python-pip

Теперь:

pip install uwsgi

**Листинг 2. Урок 1. Коды к уроку/fwsgi.py**

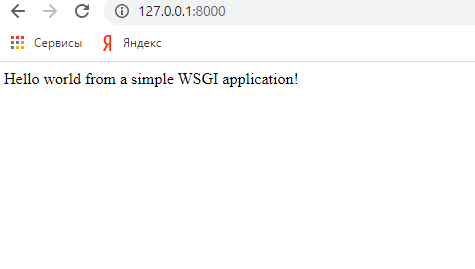
|  |
| --- |
| **def** application(environ, start\_response):  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'Hello world from a simple WSGI application!'**] |

Запускаем командой:

*uwsgi --http :8000 --wsgi-file fwsgi.py*

Заходим в браузер: <http://127.0.0.1:8000/>

Получаем:



Вам не кажется, что чего-то не хватает? А где же наш веб-сервер? Ведь мы не установили ни Apache, ни Nginx. На самом деле его роль выполняет сама библиотека uWSGI.

Обратимся к теории:

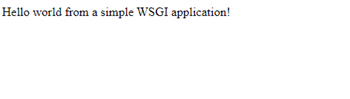
«uWSGI — веб-сервер и сервер веб-приложений, первоначально реализованный для запуска приложений Python через протокол [WSGI](https://ru.wikipedia.org/wiki/WSGI) (и его бинарный вариант uwsgi — отсюда название). Версия 2.0 поддерживает также запуск веб-приложений [Lua](https://ru.wikipedia.org/wiki/Lua), [Perl](https://ru.wikipedia.org/wiki/Perl), [Ruby](https://ru.wikipedia.org/wiki/Ruby) и других.

Может работать как самостоятельный веб-сервер, так и в интеграции с [Apache HTTP Server](https://ru.wikipedia.org/wiki/Apache_HTTP_Server), [Nginx](https://ru.wikipedia.org/wiki/Nginx), [Cherokee HTTP Server](https://ru.wikipedia.org/wiki/Cherokee_HTTP_Server) и другими.»

Идем далее!

У нас, кстати, все адреса пока обрабатываются одним обработчиком.

Т.е. на какой бы адрес мы ни указывали: <http://127.0.0.1:8000/index/>, <http://127.0.0.1:8000/about/> и т.д., мы будем получать один и тот же ответ:



Но давайте представим, как было бы хорошо, как в случае с Джанго тестировать работу проекта с помощью встроенного легковесного сервера и запускать непосредственно из PyCharm.

Такая возможность у нас есть и обеспечивается она средствами библиотеки wsgiref, которая обеспечивает нам всю необходимую инфраструктуру, в том числе возможности WSGI-коннектора и веб-сервера.

**Листинг 3. Урок 1. Коды к уроку/fwsgi\_2.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** application(environ, start\_response):  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'Hello world from a simple WSGI application!'**]   **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

*Реализуем отображение нескольких страниц*

Пусть у нас будет главная страница и страница about.

Еще раз вспомним. У нас есть **environ**. Это все данные, которые приходят сначала в Nginx, потом в Gunicorn и потом к нам в виде словаря.

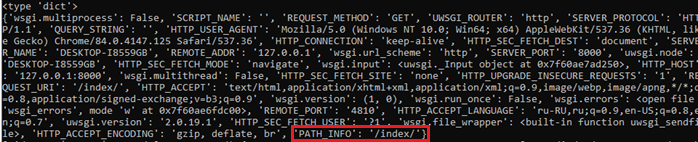
Давайте посмотрим, что у нас в этом словаре.

**Листинг 4. Урок 1. Коды к уроку/fwsgi\_3.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** application(environ, start\_response):  print(type(environ))  print(environ)  path = environ[**'PATH\_INFO'**]  **if** path == **'/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'Index'**]  **elif** path == **'/abc/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'ABC'**]  **else**:  start\_response(**'404 NOT FOUND'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'404 Not Found'**]   **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Получаем:

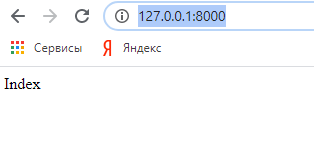


Введем в адресной строке браузера: <http://127.0.0.1:8000/index>.

Нас интересует параметр PATH\_INFO.

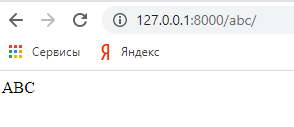
Переходим -> <http://127.0.0.1:8000/>

Получаем:



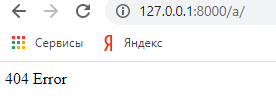
Переходим -> <http://127.0.0.1:8000/abc/>

Получаем:



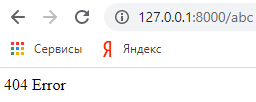
Переходим -> <http://127.0.0.1:8000/a/>

Получаем:



Обратите внимание, как важен слеш!

Переходим -> <http://127.0.0.1:8000/abc>



*Реализуем маршрутизацию как в Джанго*

Не зря мы уже изучили паттерны PC и FC.

Теперь нам нужно реализовать фреймворк так, чтобы можно было писать модели отдельно, вьюшки отдельно, url-привязки, шаблоны отдельно и т.д.

Начнем с PC.

О чем нам говорит PC?

О том, что у нас есть некоторые адреса и для каждого адреса мы делаем свою вьюшку.

**Листинг 4. Урок 1. Коды к уроку/fwsgi\_4.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   *# page controller* **def** index\_view(request):  print(request)  *# возвращаем тело ответа в виде списка из bite* **return '200 OK'**, [**b'Index'**]   **def** abc\_view(request):  print(request)  **return '200 OK'**, [**b'ABC'**]   **def** not\_found\_404\_view(request):  print(request)  **return '404 WHAT'**, [**b'404 PAGE Not Found'**]   routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view, }   **def** application(environ, start\_response):  path = environ[**'PATH\_INFO'**]  **if** path == **'/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'Index'**]  **elif** path == **'/abc/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'ABC'**]  **else**:  start\_response(**'404 Not Found'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'404 Not Found'**]   **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Теперь у нас есть вьюшки:

**def** index\_view(request):  
 print(request)  
 *# возвращаем тело ответа в виде списка из bite* **return '200 OK'**, [**b'Index'**]  
  
  
**def** abc\_view(request):  
 print(request)  
 **return '200 OK'**, [**b'ABC'**]  
  
  
**def** not\_found\_404\_view(request):  
 print(request)  
 **return '404 WHAT'**, [**b'404 PAGE Not Found'**]

И привязки их к адресам:

|  |
| --- |
| routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view, } |

Теперь нам нужно как-то передать **routes** в **application**! Можно использовать глобальную переменную **routes** в **application**. Но тогда наш фреймворк полностью свяжется с этой переменной и будет от нее зависим. Не годится!

Есть вариант передать **routes** в качестве аргумента функции **application**. Но это нельзя делать, т.к. **application** по стандарту PEP-3333 принимает только два параметра.

Поскольку приложением у нас должен быть callable-объект, то для решения проблемы мы вместо функции **application** сделаем класс **Application** и перегрузим в нем метод \_\_call\_\_().

Экземпляр такого класса можно будет вызвать как функцию. Поэтому объект называется callable-объект.

А параметр **routes** мы передадим в \_\_init\_\_().

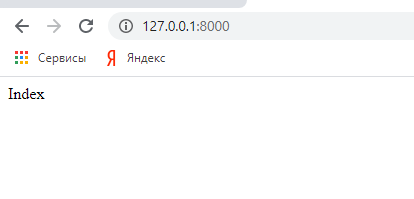
При этом выполним и требования PEP-3333, т.е. в метод \_\_call\_\_() у нас будут передаваться требуемые два параметра: environ, start\_response.

**Листинг 5. Урок 1. Коды к уроку/fwsgi\_5.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   *# page controller* **def** index\_view(request):  print(request)  **return '200 OK'**, [**b'Index'**]   **def** abc\_view(request):  print(request)  **return '200 OK'**, [**b'ABC'**]   **def** not\_found\_404\_view(request):  print(request)  **return '404 WHAT'**, [**b'404 PAGE Not Found'**]   routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view, }   **class** Application:   **def** \_\_init\_\_(self, routes):  self.routes = routes   **def** \_\_call\_\_(self, environ, start\_response):   path = environ[**'PATH\_INFO'**]  **if** path == **'/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'Index'**]  **elif** path == **'/abc/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'ABC'**]  **else**:  start\_response(**'404 Not Found'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'404 Not Found'**]   application = Application(routes)  **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Снова запускаем.

Результат такой же!



*Дорабатываем маршрутизацию*

В предыдущем листинге у нас есть один большой недостаток. Многочисленные ветвления (if-ы). Если мы захотим добавить функционал для новой страницы, то придется добавлять очередную ветвь. Плюс мы пока не задействовали наши «вьюшки».

**Листинг 6. Урок 1. Коды к уроку/fwsgi\_6.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** index\_view():  **return '200 OK'**, [**b'Index'**]   **def** abc\_view():  **return '200 OK'**, [**b'ABC'**]   **def** not\_found\_404\_view():  **return '404 WHAT'**, [**b'404 PAGE Not Found'**]   routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view, }   **class** Application:   **def** \_\_init\_\_(self, routes):  self.routes = routes   **def** \_\_call\_\_(self, environ, start\_response):   path = environ[**'PATH\_INFO'**]  **if** path **in** self.routes:  view = self.routes[path]  **else**:  view = not\_found\_404\_view  code, body = view()  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  **return** body   application = Application(routes)  **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Будем проверять, есть ли указанный путь в **routes**:

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

Если путь есть, то мы извлекаем из словаря соответствующую функцию и вызываем ее:

view = self.routes[path]

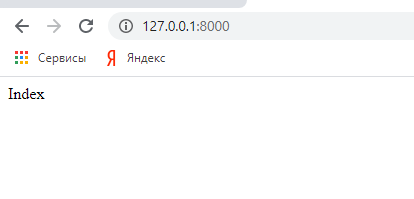
code, body = view()

Она возвращает кортеж, состоящий из кода ответа и тела ответа.

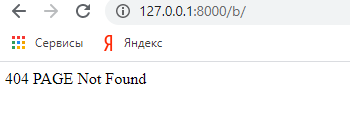
Если же пути в **routes** нет, то вызываем not\_found\_404\_view.

Снова запускаем.

Результат такой же!



Несуществующий путь также обработается.



Итак, мы в простейшем варианте реализовали логику PC.

Теперь мы можем создавать новый слой контроллера (новую «вьюшку»), и он абсолютно независим от **Application**.

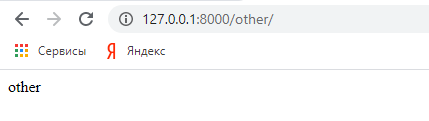
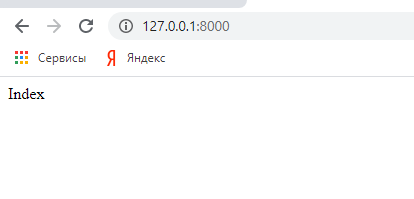
Здесь мы можем легко сделать «вьюшку» и на базе объекта класса (как и функция он с перегрузкой метода \_\_call\_\_() имеет статус callable-объекта и может быть вызван).

**Листинг 7. Урок 1. Коды к уроку/fwsgi\_7.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** index\_view():  **return '200 OK'**, [**b'Index'**]   **def** abc\_view():  **return '200 OK'**, [**b'ABC'**]   **def** not\_found\_404\_view():  **return '404 WHAT'**, [**b'404 PAGE Not Found'**]   **class** Other:   **def** \_\_call\_\_(self):  **return '200 OK'**, [**b'other'**]   routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view,  **'/other/'**: Other() }   **class** Application:   **def** \_\_init\_\_(self, routes):  self.routes = routes   **def** \_\_call\_\_(self, environ, start\_response):   path = environ[**'PATH\_INFO'**]  **if** path **in** self.routes:  view = self.routes[path]  **else**:  view = not\_found\_404\_view  code, body = view()  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  **return** body   application = Application(routes)  **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Снова запускаем.

Результат такой же!



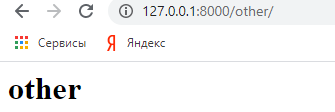
Можно также сделать, что возвращаться будет не строка, а html-страница.

**Листинг 8. Урок 1. Коды к уроку/fwsgi\_8.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** index\_view():  **return '200 OK'**, [**b'Index'**]   **def** abc\_view():  **return '200 OK'**, [**b'ABC'**]   **def** not\_found\_404\_view():  **return '404 WHAT'**, [**b'404 PAGE Not Found'**]   **class** Other:   **def** \_\_call\_\_(self):  **return '200 OK'**, [**b'<h1>other</h1>'**]   routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view,  **'/other/'**: Other() }   **class** Application:   **def** \_\_init\_\_(self, routes):  self.routes = routes   **def** \_\_call\_\_(self, environ, start\_response):   path = environ[**'PATH\_INFO'**]  **if** path **in** self.routes:  view = self.routes[path]  **else**:  view = not\_found\_404\_view  code, body = view()  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  **return** body   application = Application(routes)  **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Снова запускаем.

Получаем:



*Паттерн Front Controller*

Постепенно подбираемся к добавлению HTML-шаблонов. Но пока переходим к реализации FC. В чем сложность? Нам придется в каждый контроллер пробросить объект некоторого запроса (тот самый **request** в Django).

А зачем нам вообще может понадобиться FC? Он позволяет добавить параметр в запрос и этот запрос получат одновременно все «вьюшки».

**Листинг 9. Урок 1. Коды к уроку/fwsgi\_9.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** index\_view(request):  print(request)  **return '200 OK'**, [**b'Index'**]   **def** abc\_view(request):  print(request)  **return '200 OK'**, [**b'ABC'**]   **def** not\_found\_404\_view(request):  print(request)  **return '404 WHAT'**, [**b'404 PAGE Not Found'**]   **class** Other:   **def** \_\_call\_\_(self, request):  print(request)  **return '200 OK'**, [**b'<h1>other</h1>'**]   routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view,  **'/other/'**: Other() }   **def** secret\_front(request):  request[**'secret'**] = **'some secret'   def** other\_front(request):  request[**'key'**] = **'key'** fronts = [secret\_front, other\_front]   **class** Application:   **def** \_\_init\_\_(self, routes, fronts):  self.routes = routes  self.fronts = fronts   **def** \_\_call\_\_(self, environ, start\_response):   path = environ[**'PATH\_INFO'**]  **if** path **in** self.routes:  view = self.routes[path]  **else**:  view = not\_found\_404\_view  request = {}  *# front controller* **for** front **in** self.fronts:  front(request)  print(request)  code, body = view(request)  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  **return** body   application = Application(routes, fronts)  **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Мы создали пустой словарь:

request = {}

Пропустили его через функции фронт-контроллера:

**for** front **in** self.fronts:  
 front(request)

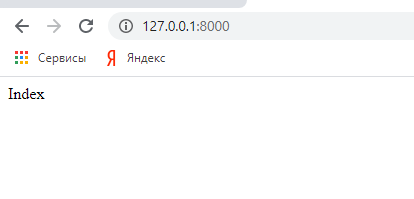
**def** secret\_front(request):  
 request[**'secret'**] = **'some secret'  
  
  
def** other\_front(request):  
 request[**'key'**] = **'key'**

Получившийся модифицированный словарь-запрос получат все наши «вьюшки»:

code, body = view(request)

С параметрами словаря «вьюшки» могут выполнить какие-либо операции.

Снова запускаем.

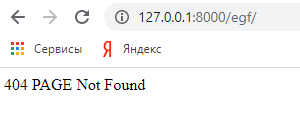


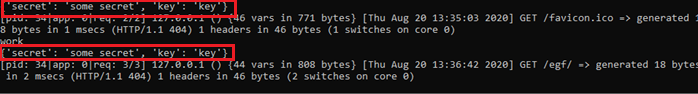
Получаем:



Теперь запустим с ошибкой.

Получаем те же самые данные.





Теперь эти параметры мы можем использовать во всех «вьюшках».

Теперь мы можем подключать слои и отключать их – создавать и вносить нужные функции фронт-контроллера в список.

**def** secret\_front(request):  
 request[**'secret'**] = **'some secret'  
  
  
def** other\_front(request):  
 request[**'key'**] = **'key'**fronts = [secret\_front, other\_front]

Теперь приложение можно расширять в ширину, добавляя урлы и обработчики, и в высоту, добавляя новые слои для всех запросов.

*Подключаем шаблонизатор*

Нужно понимать, что если страница объемная, то делать вот так:

return **'200 OK'**, [b'<h1>other</h1>']

Уже будет не очень хорошо.

Будет нужна или большая строка, или придется где-то эту страницу хранить.

Нужен шаблон для страницы и диспетчер шаблонизации – шаблонизатор.

Свой шаблонизатор писать не будем. Воспользуемся Jinja2.

Установим jinja2 командой:

*pip install jinja2*

**Листинг 9. Урок 1. Коды к уроку/templator.py**

|  |
| --- |
| *""" Используем шаблонизатор jinja2 """* **from** jinja2 **import** Template   **def** render(template\_name, \*\*kwargs):  *"""  Минимальный пример работы с шаблонизатором* **:param** *template\_name: имя шаблона* **:param** *kwargs: параметры для передачи в шаблон* **:return***:  """  # Открываем шаблон по имени* **with** open(template\_name, encoding=**'utf-8'**) **as** f:  *# Читаем* template = Template(f.read())  *# рендерим шаблон с параметрами* **return** template.render(\*\*kwargs)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  *# Пример использования* output\_test = render(**'authors.html'**,  object\_list=[{**'name'**: **'Leo'**}, {**'name'**: **'Kate'**}])  print(output\_test) |

Для рендеринга шаблона удобно реализовать собственную функцию:

**def** render(template\_name, \*\*kwargs):

Передаем в эту функцию имя шаблона.

\*\*kwargs – это параметры, которые мы можем передать в шаблон. Это контекст или еще по-другому динамически передаваемые в шаблон данные. Вы же помните это по курсу Джанго.

Далее открываем шаблон.

**with** open(template\_name, encoding=**'utf-8'**) **as** f:

Здесь нам нужен класс-конструктор Template из Jinja2.

template = Template(f.read())

Выполняем рендеринг шаблона с передачей в него контекста.

**return** template.render(\*\*kwargs)

Представленный вариант отлично сработает, если шаблон «лежит» в той же папке, где и файл tempalator.py.

Рассмотрим на примере шаблона **authors.html.**

**Листинг 10. Урок 1. Коды к уроку/authors.html**

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Authors</**title**> </**head**> <**body**>  <**h1**>Авторы</**h1**>  <**ul**>  {{object\_list[0]}}  {{object\_list[0]['name']}}   {{object\_list.0}}  {{object\_list.0.name}}   {% for author in object\_list %}  <**li**>{{author.name}}</**li**>  {% endfor %}  </**ul**> </**body**> </**html**> |

Запустим файл **templator**:

