# Введение

*СЛАЙД 1-2*

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. Цель курса в освоении фундаментальных паттернов, т.е. некоторых шаблонов реализации программных продуктов, а также в изучении возможностей их применения на примере своего проекта.

Речь идет прежде всего о паттернах, применяемых в веб-программировании и по итогам курса, мы напишем свой оригинальный WSGI-фреймворк (не забудьте придумать для него название), упрощенную копию фреймворков Django и Flask. Не обойдем вниманием и шаблонизатор. Возможно, попытаемся реализовать свою ORM. Курс у нас обновленный, он видоизменился и теперь у нас 9 уроков вместо 8. Добавился еще урок по микросервисам.

*СЛАЙД 3*

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) и постараемся реализовать ORM.

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

*СЛАЙД 4*

Что это нам даст?

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.

*СЛАЙД 5*

На сегодняшнем уроке мы вспомним (или выучим, если вы еще не встречались с этим ранее), в чем смысл концепции 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.

Page Controller и Front Controller – очень важные понятия, не все о них знают.

*СЛАЙД 6*

Вводная часть завершена, переходит непосредственно к сути урока – паттернам веб-представления данных.

Нам нужно понять, как, опираясь на паттерны, создавать свои веб-приложения с логикой, интерфейсом и базой данных.

*СЛАЙД 7*

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:

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

**Контроллер**. Обработка запросов.

**View**. Дизайн (оформление) приложения. То, что видит пользователь, но необязательно это HTML.

Контроллер знает про Модель 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. Модели – это наши классы – шаблоны таблиц БД, унаследованные от model.models. View – это наши «вьюхи», 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.

*СЛАЙД 8*

Что касается Django (да и Flask тоже), то там есть так называемые Урлы (или Роуты, маршруты).

*СЛАЙД 9*

Плюсы и минусы 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.

*СЛАЙД 10*

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

*СЛАЙД 11*

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.

*СЛАЙД 12*

MVC – фреймворки. Самой собой, это Django, на Python. Во Фласке это тоже есть, но там нет моделей. Можно подключить SQLAlchemy, шаблонизатор там идет на 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.

*СЛАЙД 13*

## Page Controller

Это, по сути, то, с чем вы работали на Django. Когда говорят про MVC, обычно представляют себе именно Page Controller.

*СЛАЙД 14*

На сайте есть адреса. Пользователи отправляют на них запросы. К адресам привязаны «вьюхи» (функции или классы). По сути, этот паттерн – разновидность MVC.

*СЛАЙД 15*

**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 и смотрим, какая в ней логика.

Отделяет логику от представления. Мы делаем рЕндеринг шаблона и отдаем данные.

Подмножество MVC.

*СЛАЙД 16*

В представленных продуктах реализован Page Controller.

*СЛАЙД 17-18*

## Front Controller

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

*СЛАЙД 19*

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.

Но есть и минусы:

Мы не можем писать логику для каждого адреса в отдельности.

*СЛАЙД 20-21*

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.

*СЛАЙД 22*

Уделить внимание оптимизации кода. В Middleware можно реализовывать безопасность, делать мониторинг, делать оптимизацию.

*СЛАЙД 23*

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

Берем Nginx, вешаем к нему Gunicorn, Gunicorn «смотрит» приложение, и они в связке работают.

У нас есть сервера Nginx и, например, 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 других 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 нам данные.

Т.е. пользователь «стучится», сервера передают данные, которые кто-то может читать. Обычно, эти данные – просто набор байт.

У нас есть приложение на Python, но данные приходят в байтах. Нужен обработчик, который будет понимать WSGI-ответ.

Нам нужно превратить ответ в нормальный, читабельный вид, например, в тип dict.

Я хочу работать с dict, но данные приходят в байтах. Мне нужно байты прочитать в словарь.

Этим и занимается Gunicorn. Или другая популярная библиотека uWSGI.

**ИТОГО:** Nginx дал данные, которые через протокол WSGI приняты и далее с помощью Gunicorn прочитаны в словарь, а словарь отдан приложению на Python.

И в обратную сторону. Приложение на Python должно отдать ответ в Gunicorn. И вот эти библоиотеки-коннекторы (Gunicorn и uWSGI) передают ответ Nginx-у.

*СЛАЙД 24*

*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.*

В следующем листинге представлено начало 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 класс с методом call.

**Листинг 1. 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!'] |

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 коды ответов.

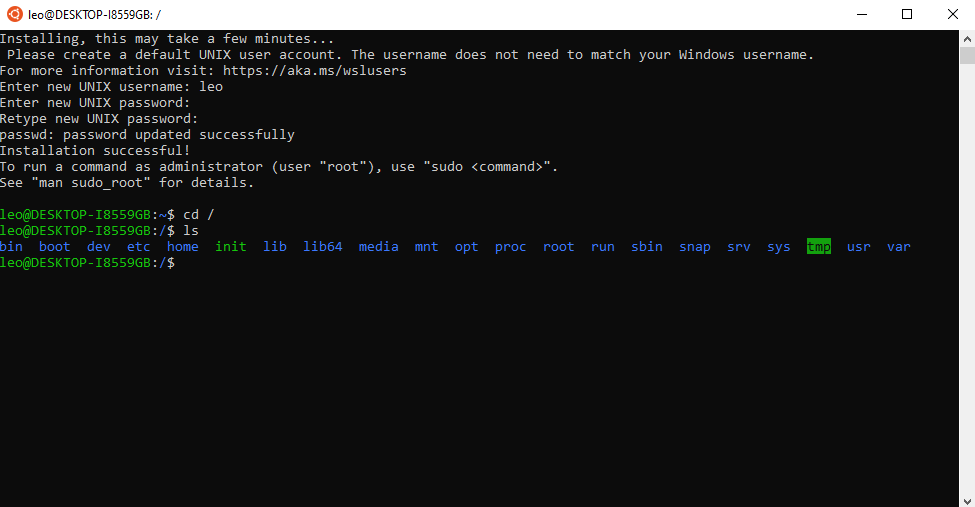
Теперь мы можем соединить этот callable-объект с gunicorn-ом или uwsgi-ем.

Мы запускаем gunicorn, указываем приложение (def application), которое будет работать с gunicorn-ом. И таким образом, будем уже взаимодействовать с веб-сервером.

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.

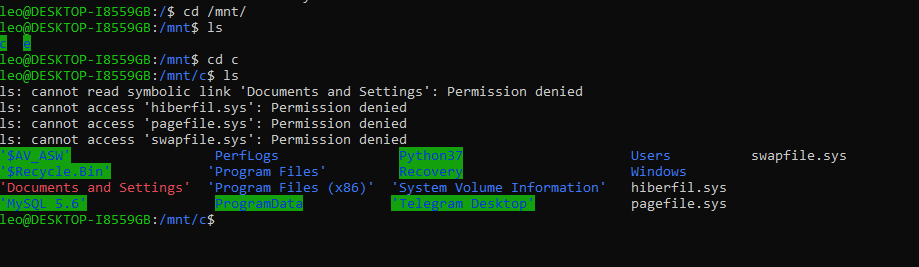
Есть решение для Windows 10!

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

**

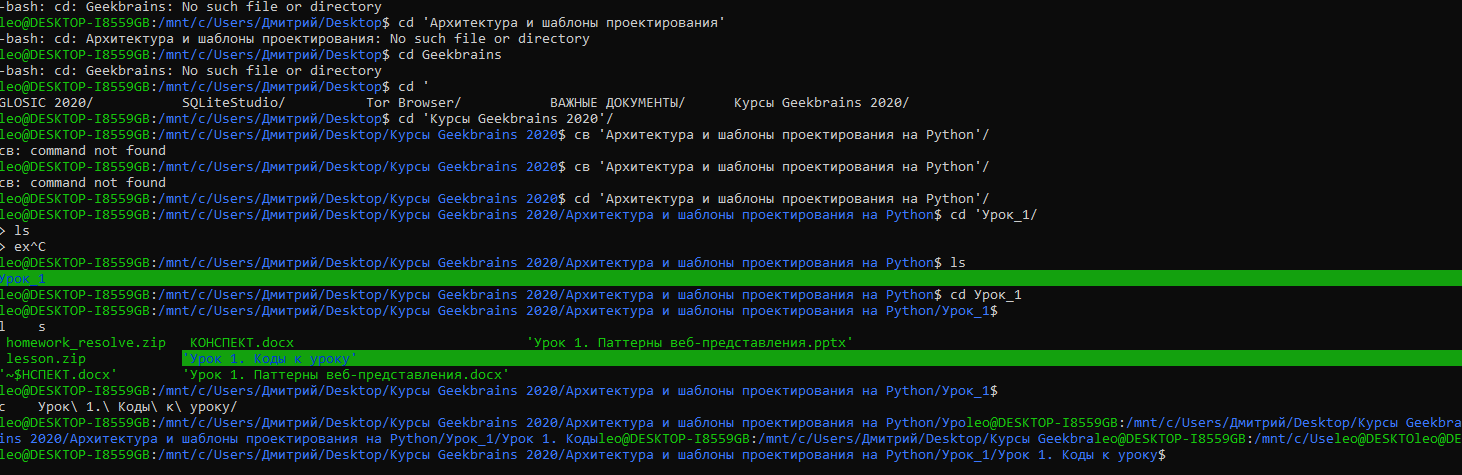
Очень похоже на файловую систему linux, не правда ли? Кстати при возникновении проблем с установкой воспользуйтесь инструкцией - <http://windowsbulletin.com/ru/fix-wslregisterdistribution-failed-with-error-0x8007019e-and-0x8000000d/>.

Но здесь мы можем перейти как будто перемещаемся по файловой системе Windows.

**

Не нужно «шарить» папки, можно обращаться напрямую.

Приступаем к запуску!



Файл simple\_wsgi.py не трогаем. Он будет шаблоном.

Запускать будем файл fwsgi.py

**Листинг 2. fwsgi.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 начала через uwsgi. Но его 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

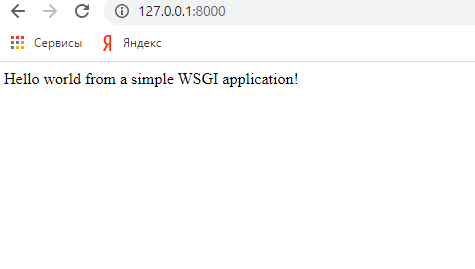
pip install uwsgi

Для начала запустим через *uwsgi --http :8000 --wsgi-file fwsgi.py*

Итак, теперь наше приложение висит с помощью wsgi.

Теперь можем потестить. Для этого 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.

“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" \o "Lua), [Perl](https://ru.wikipedia.org/wiki/Perl" \o "Perl), [Ruby](https://ru.wikipedia.org/wiki/Ruby" \o "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" \o "Apache HTTP Server), [Nginx](https://ru.wikipedia.org/wiki/Nginx" \o "Nginx), [Cherokee HTTP Server](https://ru.wikipedia.org/wiki/Cherokee_HTTP_Server" \o "Cherokee HTTP Server) и другими.”

Идем далее!

Теперь наше приложение может работать как wsgi. Теперь мы можем его дописывать, создавая полезные вещи.

Попробуем сделать фреймворк, который будет отрисовывать две страницы.

У нас кстати все адреса пока обрабатываются одним ответом, например, адрес <http://127.0.0.1:8000/index/>

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.

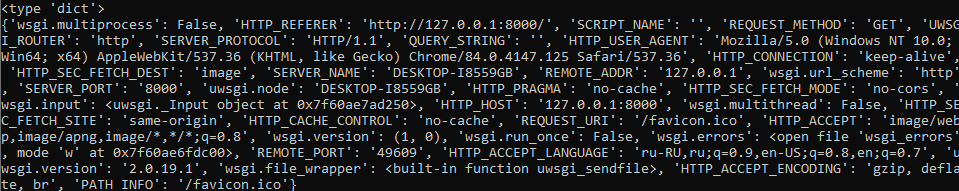
**Листинг 2. fwsgi\_2.py**

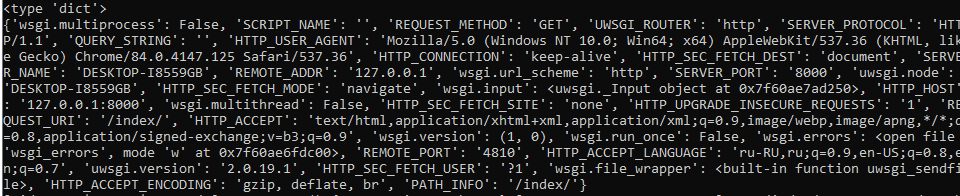
|  |
| --- |
| def application(environ, start\_response):  print(type(environ))  print(environ)  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  return [b'Hello world from a simple WSGI application!'] |

Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_2.py*

Получаем:



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

Нас интересует параметр PATH\_INFO. Поработаем с ним.

**Листинг 3. fwsgi\_3.py**

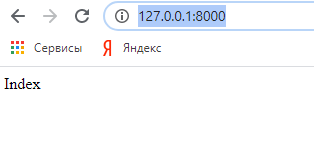
|  |
| --- |
| def application(environ, start\_response):  print(type(environ))  print(environ)  path = environ[**'PATH\_INFO'**]  if path == **'/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  return [b'Index']  elif path == **'/abc/'**:  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  return [b'ABC']  else:  start\_response(**'404 NOT FOUND'**, [(**'Content-Type'**, **'text/html'**)])  return [b'404 Error'] |

Снова запускаем.

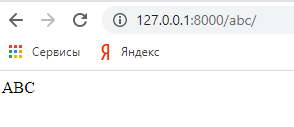
*uwsgi --http :8000 --wsgi-file fwsgi\_3.py*

Переходим -> <http://127.0.0.1:8000/>

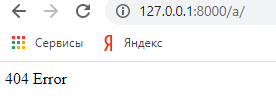
Получаем:



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

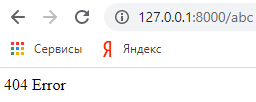


Переходим -> <http://127.0.0.1:8000/a/>



Обратите внимание, как важен слэш!

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



Обратите внимание, что нам важен правильный протокол кодов!

return [b'404 Not Found']

<https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP>

Сделаем из всего этого упрощенную версию WSGI-фреймворка, например Фласка или Джанго.

Не зря мы уже изучили паттерны PC и FC.

Теперь нам нужно реализовать фреймворк так, чтобы можно было писать модели отдельно, вьюхи отдельно, 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. fwsgi\_4.py**

|  |
| --- |
| *# 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):  return **'200 OK'**, [b'ABC']   def not\_found\_404\_view(request):  print(request)  return **'404 WHAT'**, [b'404 PAGE Not Found'] |

Теперь у нас есть вьюшки и нужны маршрутизация (урлы или роуты).

**Листинг 4. fwsgi\_4.py**

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

Для страницы 404 маршрут пока не делаем.

Теперь нам нужно как-то передать роуты в приложение! Можно использовать глобальную переменную routes в application. Но тогда наш фреймворк полностью свяжется с этой переменной.

Есть вариант передать routes в ф-цию application. Но это нельзя делать, т.к. application принимает только два параметра.

ДЛЯ РЕШЕНИЯ ПРОБЛЕМЫ МЫ ВОСПОЛЬЗЕМСЯ КЛАССОМ С МЕТОДОМ CALL!

Тогда экземпляр класса затем можно будет вызвать как функцию. Поэтому объект называется callable-объект.

А необходимые параметры мы передадим в \_\_init\_\_.

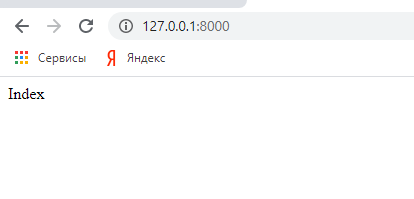
**Листинг 5. fwsgi\_5.py**

|  |
| --- |
| *# 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):  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):  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']  application = Application(routes) |

Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_5.py*

Результат такой же!



Продолжим писать код. Проверим, есть ли путь в роутах!

Если путь есть, то мы достаем из словаря соответствующую функцию и вызываем ее. Она возвращает кортеж, состоящий из кода ответа и тела ответа.

Если же пути в ротуах нет, то вызывает not\_found\_404\_view.

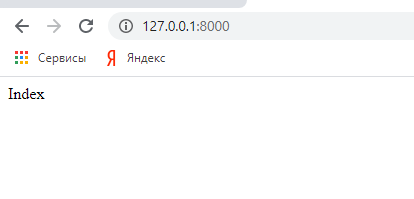
**Листинг 6. fwsgi\_6.py**

|  |
| --- |
| *# page controller* 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):  print(**'work'**)  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) |

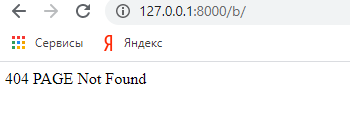
Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_6.py*

Результат такой же!



Несуществующий путь также обработается.



Итак, мы в простейшем варианте реализовали логику PC.

Теперь мы можем создавать новый слой контроллера (новую вьюху), и он абсолютно независим от Application.

Здесь мы можем легко сделать вьюшку и на классах!

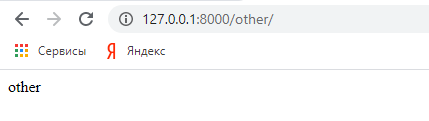
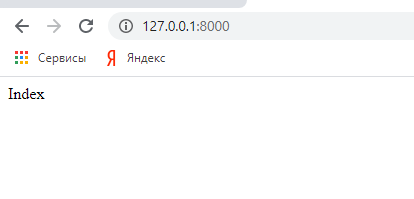
**Листинг 7. fwsgi\_7.py**

|  |
| --- |
| *# page controller* 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):  print(**'work'**)  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) |

Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_7.py*

Результат такой же!



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

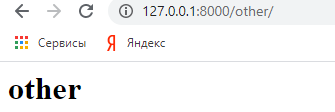
**Листинг 8. fwsgi\_8.py**

|  |
| --- |
| *# page controller* 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):  print(**'work'**)  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) |

Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_8.py*

Получаем:



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

А зачем нам вообще может понадобиться FC? Он позволяет добавить параметр одновременно во все запросы.

Добавим объект request во все вьюхи!

Когда мы вызываем вью в Application, должен быть некоторый запрос. Пусть это будет пустой словарь, в который потом положим некоторый объект.

Создали пустой словарь и во вьюшку его передали (во все вьюшки он будет приходить).

Объект запроса можно и вывести (запринтить).

**Листинг 9. fwsgi\_9.py**

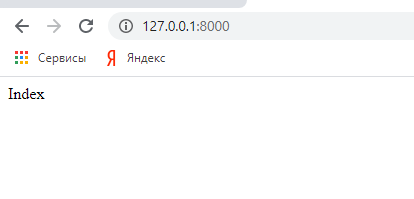
|  |
| --- |
| *# page controller* def index\_view(request):  print(request)  return **'200 OK'**, [b'Index']   def abc\_view(request):  print(request)  return **'200 OK'**, [b'ABC']   def not\_found\_404\_view(request):  print(request)  return **'404 WHAT'**, [b'404 PAGE Not Found']  class Other:   def \_\_call\_\_(self, request):  print(request)  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):  print(**'work'**)  path = environ[**'PATH\_INFO'**]  if path in self.routes:  view = self.routes[path]  else:  view = not\_found\_404\_view  request = {}  code, body = view(request)  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  return body   application = Application(routes) |

Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_9.py*

Получаем:

Результат такой же!





Зачем нужен словарь? Мы будем прогонять данные через FC. Там есть слои. Каждый слой сможет добавить что-то в этот словарь. И либо мы подключаем слой, либо не подключаем.

*СЛАЙД 26*

Реализуем слой в виде ф-ции secret\_front. В Джанго это Middleware. Задача secret\_front что-то добавить в запрос, т.е. в наш словарь!

Можно добавить и еще один некоторый абстрактный FC, например, other\_front.

Что делать дальше с этими FC? FC – это слои и мы можем их хранить не в словаре, а в списке!

Формируем список контроллеров и передает в \_\_init\_\_ класса Application.

Далее, перед теми, как передать запрос во view, мы прогоняем все запросы через наши FC, используя цикл.

Не забываем передать fronts в класс Application.

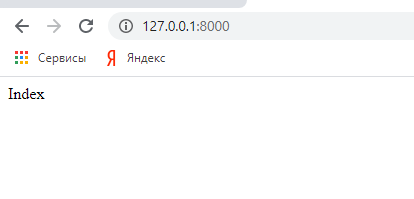
**Листинг 10. fwsgi\_10.py**

|  |
| --- |
| *# 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']  class Other:   def \_\_call\_\_(self, request):  print(request)  return **'200 OK'**, [b'<h1>other</h1>']  routes = {  **'/'**: index\_view,  **'/abc/'**: abc\_view,  **'/other/'**: Other() }  *# Fron controllers* 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):  print(**'work'**)  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)  code, body = view(request)  start\_response(code, [(**'Content-Type'**, **'text/html'**)])  return body   application = Application(routes, fronts) |

Теперь при любом запросе во все запросы будут передаваться эти данные словаре.

Снова запускаем.

*uwsgi --http :8000 --wsgi-file fwsgi\_10.py*

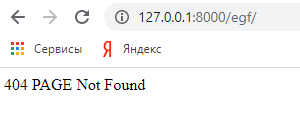


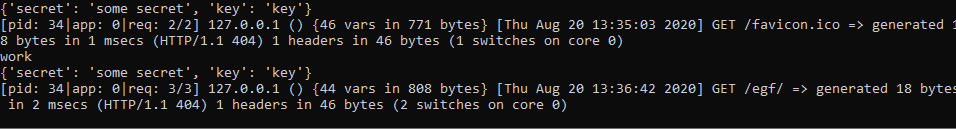
Получаем:



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 – вносить нужные FC в список.

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

Можно также сделать, что FC будет добавлять что-то не только в запрос, но и в ответ.

Переходим к шаблонам.

Нужно понимать, что если страница большая, 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, в который мы сможем вставлять теги. В джанго это называется шаблонизатором.

Свой шаблонизатор писать не будем. Воспользуемся Jinja2. Он есть и отдельно, а не только во фласке.

Установим jinja2 командой:

*pip install jinja2*

**Листинг 11. 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) |

Рендеринг шаблона удобно «положить» в функцию.

Передаем в эту функцию имя шаблона.

\*\*kwargs – это параметры, которые мы можем передать как переменные в шаблон.

Далее открываем шаблон. Здесь нам нужен класс Template из jinja2.

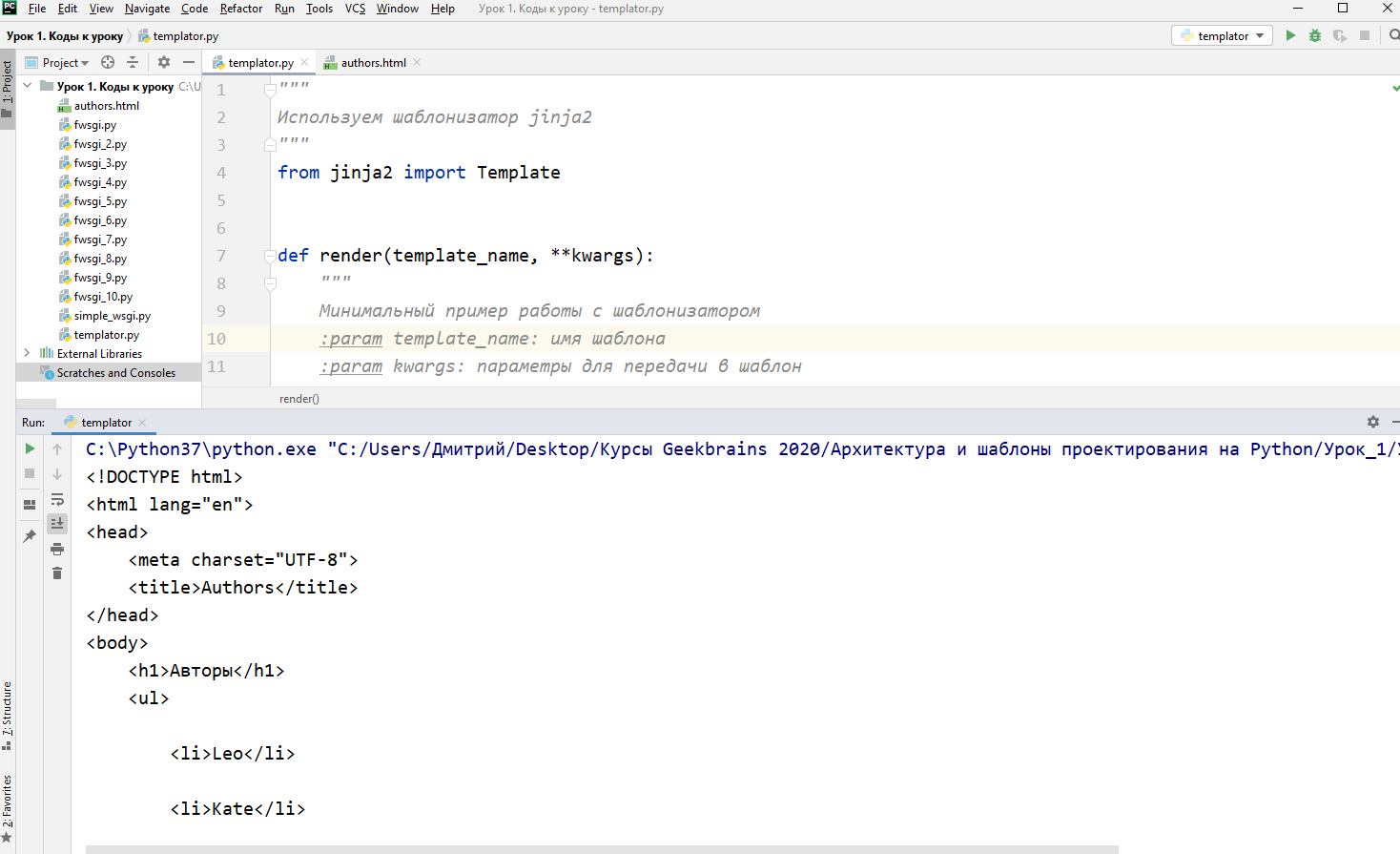
Представленный вариант отлично сработает, если шаблон «лежит» в той же папке, где и файл tempalator.py.

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

**Листинг 12. 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:



По первому уроку все!

Теперь о ДЗ, слайды 30-32.

Поговорим о сути нашего проекта.

Наш проект называется «Обучающий сайт».

Начинаем мы с создания фреймворка. Его будем делать первые три занятия. (wsgi-фреймворк).

Сегодня мы отрабатывает добавление во фреймворк PC и FC.

Далее пойдет отработка пост-запроса, и всего того, что понадобится для написания реального сайта.

Далее будем работать с бизнес-логикой и делать обучающий сайт на базе этого фреймворка.

Это будет небольшой сайт, где можно добавлять курсы и учить всему, чему вы хотите.

В этом контексте мы потренируемся в реализации классических паттернов (три занятия).

Далее займемся реализацией БД (1 занятие, пишем сами или возьмем алхимию).

Далее антипаттерны (1 занятие).

Микросервисы (1 занятие).

Таким образом, почти все пишем сами, кроме шаблонизатора. Возьмем Jinja2, как стороннюю библиотеку.

Шаги в ДЗ, слайды 33

Вы можете сделать и два репозитория – один для фреймворка, второй – для сайта.

Можно даже придумать название своему фреймворку и как пакет выложить на PyPi.