

Урок 5. Учим фреймворк выполнять привязку URL-шаблонов к обработчикам

ОГЛАВЛЕНИЕ

[Что такое маршрутизация 3](#_Toc75781020)

[КАК реализовать маршрутизацию через декоратор 4](#_Toc75781021)

[Заключение 10](#_Toc75781022)

[КЛЮЧЕВЫЕ ВЫВОДЫ 10](#_Toc75781023)



# Что такое маршрутизация

Вспомним начало нашего курса. Еще на первом уроке мы отметили, что планируем создать фреймворк-аналог известных продуктов – Django и Flask. По итогам четвертого урока наш фреймворк стал похож в большей степени на Django. Например, мы создали файл **urls.py**, в котором определили привязки URL-адресов и контроллеров, обрабатывающих запросы по этим адресам. В проектах на Django также присутствую такие файлы **urls.py**.

Во фреймворке Flask применяется несколько иной подход, под названием маршрутизация, при которой файл **urls.py** отсутствует, и мы указываем какой путь обрабатывает каждый контроллер непосредственно в файле с контроллерами. При этом применяется механизм декораторов, когда каждая функция или класс-контроллер помещаются в соответствующую обертку-декоратор. Такие декораторы называются «роутами» или «маршрутами». Это выглядит, например, так:

@app.route(**'/sale/'**, methods=[**'POST'**])  
**def** sale():  
 data = request.form  
 phone = data[**'phone'**]

Можем ли мы в нашем проекте уйти от подхода, существующего в Django и заменить его на подход, применяющийся во Flask? Тогда файл **urls.py** нам станет ненужным. Но придется подготовить собственный декоратор, обеспечивающий возможности маршрутизации.

У нас еще на предыдущем уроке в файле **urls.py** был словарь, содержащий пары ключ-значение. Ключом выступал путь, а значением – объект класса-обработчик.

*# Набор привязок: путь-контроллер*routes = {  
 **'/'**: Index(),  
 **'/about/'**: About(),  
 **'/study\_programs/'**: StudyPrograms(),  
 **'/courses-list/'**: CoursesList(),  
 **'/create-course/'**: CreateCourse(),  
 **'/create-category/'**: CreateCategory(),  
 **'/category-list/'**: CategoryList()  
}

# КАК реализовать маршрутизацию через декоратор

Как нам сделать механизм, схожий с механизмом фреймворка Flask? Идея в том, что должна быть некоторая функция или класс, которая будет заполнять аналогичный словарь. Т.е. не разработчик вручную это будет делать, а некоторая функция или класс. Мы сделаем ее декоратором для наших контроллеров в файле **views.py**.

Мы будем декорировать каждый контроллер и по этому контроллеру будет добавляться очередной элемент в словарь. И когда при запуске приложения мы перейдем на какую-то страницу, будет извлечен нужный класс-контроллер и будет выполнена обработка запроса.

Чтобы это все сделать, нам нужно вспомнить особенности работы декораторов в Python. Декоратор – это функция или объект класса (с перегрузкой метода **\_\_call\_\_()**), выполняющие роль обертки для некоторой функции или класса. Задача декоратора – дополнить возможности функции еще каким-либо полезным функционалом. Т.е. создать декоратор – значит создать callable-объект. Пример декоратора, необходимого нашему проекту, представлен в листинге ниже.

**Листинг 1. step\_5/components/decorators.py**

|  |
| --- |
| *# Декоратор для реализации маршрутизации* **class** AppRoute:  **def** \_\_init\_\_(self, routes, url):  *"""  Сохраняем значение переданного параметра  """* self.routes = routes  self.url = url   **def** \_\_call\_\_(self, cls):  *"""  Сам декоратор  """* self.routes[self.url] = cls() |

Из листинга видно, что в качестве варианта реализации декоратора выбран ООП-подход, т.е. декоратор представлен в виде callable-объекта.

Метод **\_\_init\_\_()** принимает два важных параметра – **routes** и **url**. Первый – это словарь, который мы будем наполнять. Второй – очередной url-путь, который нужно добавить в словарь.

Теперь самое главное. Как декоратор будет наполнять словарь? В методе **\_\_call\_\_()** (он ведь нам очень нужен, т.к. объект класса-декоратора будет вызываться как функция) нам нужно прописать вот такую интересную команду:

self.routes[self.url] = cls()

Эта команда позволит создать в словаре элемент, где ключом будет путь, а значением объект класса-контроллер, который мы декорируем, например, «Курс», «Категория» и т.д.

Итак, наш декоратор готов. Теперь необходимо импортировать его в файл views.py и применить к нашим классам контроллерам. В файле views.py мы сразу создадим пустой словарь, который и будем наполнять нашими привязками.

**Листинг 2. step\_5/views.py**

|  |
| --- |
| routes = {}  *# Класс-контроллер - Главная страница* @AppRoute(routes=routes, url=**'/'**) **class** Index:  **def** \_\_call\_\_(self, request):  **return '200 OK'**, render(**'index.html'**, objects\_list=site.categories)   *# Класс-контроллер - Страница "О проекте"* @AppRoute(routes=routes, url=**'/about/'**) **class** About:  **def** \_\_call\_\_(self, request):  **return '200 OK'**, render(**'about.html'**)   *# Класс-контроллер - Страница "Расписания"* @AppRoute(routes=routes, url=**'/study\_programs/'**) **class** StudyPrograms:  **def** \_\_call\_\_(self, request):  **return '200 OK'**, render(**'study-programs.html'**, data=date.today())   *# Класс-контроллер - Страница 404* **class** NotFound404:  **def** \_\_call\_\_(self, request):  **return '404 WHAT'**, **'404 PAGE Not Found'** *# Класс-контроллер - Страница "Список курсов"* @AppRoute(routes=routes, url=**'/courses-list/'**) **class** CoursesList:  **def** \_\_call\_\_(self, request):  **try**:  category = site.find\_category\_by\_id(  int(request[**'request\_params'**][**'id'**]))  **return '200 OK'**, render(**'course\_list.html'**,  objects\_list=category.courses,  name=category.name,  id=category.id)  **except** KeyError:  **return '200 OK'**, **'No courses have been added yet'** *# Класс-контроллер - Страница "Создать курс"* @AppRoute(routes=routes, url=**'/create-course/'**) **class** CreateCourse:  category\_id = -1   **def** \_\_call\_\_(self, request):  **if** request[**'method'**] == **'POST'**:  *# метод пост* data = request[**'data'**]   name = data[**'name'**]  name = site.decode\_value(name)   category = **None  if** self.category\_id != -1:  category = site.find\_category\_by\_id(int(self.category\_id))   course = site.create\_course(**'record'**, name, category)  site.courses.append(course)   **return '200 OK'**, render(**'course\_list.html'**,  objects\_list=category.courses,  name=category.name,  id=category.id)   **else**:  **try**:  self.category\_id = int(request[**'request\_params'**][**'id'**])  category = site.find\_category\_by\_id(  int(self.category\_id))   **return '200 OK'**, render(**'create\_course.html'**,  name=category.name,  id=category.id)  **except** KeyError:  **return '200 OK'**, **'No categories have been added yet'** *# Класс-контроллер - Страница "Создать категорию"* @AppRoute(routes=routes, url=**'/create-category/'**) **class** CreateCategory:   **def** \_\_call\_\_(self, request):   **if** request[**'method'**] == **'POST'**:  *# метод пост* print(request)  data = request[**'data'**]   name = data[**'name'**]  name = site.decode\_value(name)   category\_id = data.get(**'category\_id'**)   category = **None  if** category\_id:  category = site.find\_category\_by\_id(int(category\_id))   new\_category = site.create\_category(name, category)   site.categories.append(new\_category)   **return '200 OK'**, render(**'index.html'**,  objects\_list=site.categories)  **else**:  categories = site.categories  **return '200 OK'**, render(**'create\_category.html'**,  categories=categories)   *# Класс-контроллер - Страница "Список категорий"* @AppRoute(routes=routes, url=**'/category-list/'**) **class** CategoryList:   **def** \_\_call\_\_(self, request):   **return '200 OK'**, render(**'category\_list.html'**,  objects\_list=site.categories) |

Вот так выглядит пример маршрута в нашем проекте:

@AppRoute(routes=routes, url=**'/create-category/'**)

А так выглядело объявление метода \_\_init\_\_() в декораторе:

**def** \_\_init\_\_(self, routes, url):

Т.е. декоратор, принимая url, создает для него элемент словаря с объектом класса-обработчика в качестве значения.

Но есть один важный момент. На момент вызова контроллера-обработчика наш словарь уже должен быть заполнен. Иначе этот контроллер просто не будет вызван, т.к. в словаре не будет нужного элемента ключ-значение.

Но как заполнить словарь до вызова самих контроллеров-обработчиков. Здесь нам поможет важная особенность декораторов – они исполняются еще на этапе импорта. Важно понять – не декорированные контроллеры исполняются, а именно декораторы. Т.е. мы вызываем еще не контроллеры, а вот эти выражения-декораторы:

@AppRoute(routes=routes, url=**'/create-category/'**)

@AppRoute(routes=routes, url=**'/create-course/'**)

Декораторы заполнят наш словарь элементами. И заполнят они этот словарь еще на этапе его импорта из файла **views.py**.

**Листинг 3. step\_5/run.py**

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

Вот тот самый импорт:

**from** views **import** routes

Теперь, если мы запустим проект и проверим его работу, как на уроке 4, то убедимся, что приложение работает как раньше. Ничего не изменилось, только маршрутизацию мы реализовали через декораторы-роуты.

Теперь можно смело удалить из нашего проекта файл **urls.py**. Он больше не нужен.

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

# Вид сверху на рукопожатие над деловыми документамиЗаключение

На пятом уроке мы научили наш фреймворк осуществлять формирование приязок контроллеров к адресам через декораторы. Теперь наш фреймворк стал похож на Flask. Мы больше работаем с файлом urls.py и пути, которые обрабатываются контроллерами, мы указываем непосредственно в файле views.py.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Декораторы отрабатывают на этапе импорта, а значит мы можеи использовать их для наполнения нашего файла с привязками «путь-контроллер».
* Декоратор может быть реализован в виде функции или объекта класса с перегрузкой метода \_\_call\_\_(). Поскольку в нашем проекте больший упор сделан на ООП, мы реализуем декоратор в виде объекта класса.
* Предложенный подход не влияет на итоговую работу проекта. Все реализованные на предыдущих уроках возможности полностью сохраняются.

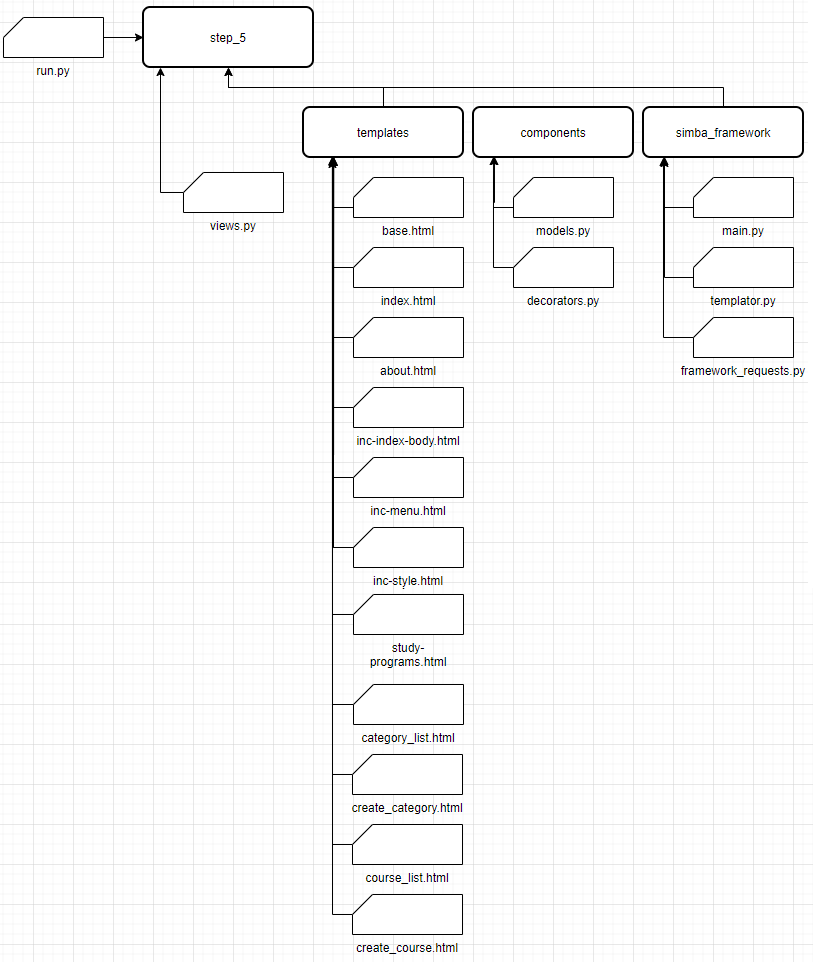
Текущая структура предложенного WSGI-фреймворка и создаваемого на его основе веб-приложения приведена на рис. 1.

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