

Урок 6. Реализуем во фреймворке возможность написания обработчиков по CBV-подходу

ОГЛАВЛЕНИЕ

[Что такое встроенные классы-контроллеры (CBV) 3](#_Toc75782499)

[Реализуем первый собственный встроенный класс-обработчик 4](#_Toc75782500)

[применяем первый собственный встроенный класс-обработчик 5](#_Toc75782501)

[Реализуем второй собственный встроенный класс-обработчик 7](#_Toc75782502)

[Применяем второй собственный встроенный класс-обработчик 8](#_Toc75782503)

[Дополняем проект необходимыми шаблонами 10](#_Toc75782504)

[проверяем работу проекта 12](#_Toc75782505)

[Заключение 15](#_Toc75782506)

[КЛЮЧЕВЫЕ ВЫВОДЫ 15](#_Toc75782507)



# Что такое встроенные классы-контроллеры (CBV)

Наш фреймворк уже умеет обрабатывать поступающие запросы, но обработчики таких запросов (контроллеры) мы пока реализовали в виде функций или объектов классов. Но вспомним, какая полезная опция существует во фреймворке Django. Это подход использования Class-based views (CBV). Это специальные служебные классы, которые имеют некоторый встроенный интерфейс. Это очень удобно, т.к. такие классы универсальны и имеют уже некоторую встроенную логику, которую мы можем использовать при написании своих контроллеров.

Ранее нам приходилось всю логику контроллеров прописывать самостоятельно. Благодаря CBV, путем наследования интерфейса данных классов, мы можем писать меньше кода, при этом реализовывать необходимую функциональность. Этот подход существует в Django и позволяет обеспечивать все компоненты механизма CRUD (create, read, update, delete), а также успешно представлять списки записей. Это встроенные классы, реализованные разработчиками Django. Можем ли мы создать для своего фреймворка аналоги таких классов? Почему нет. Реализуем для начала несколько CBV-классов.

# Реализуем первый собственный встроенный класс-обработчик

**Листинг 1. step\_6/components/cbv.py**

|  |
| --- |
| *# Класс-контроллер для простейшего рендеринга HTML-шаблонов* **class** TemplateView:  template\_name = **'template.html'   def** get\_context\_data(self):  **return** {}   **def** get\_template(self):  **return** self.template\_name   **def** render\_template\_with\_context(self):  template\_name = self.get\_template()  context = self.get\_context\_data()  **return '200 OK'**, render(template\_name, \*\*context)   **def** \_\_call\_\_(self, request):  **return** self.render\_template\_with\_context() |

Вспомним, что в Django тоже был такой встроенный класс. Он обеспечивал простейший рендеринг шаблона с передачей в него контекста. От этого класса будут наследоваться классы-контроллеры нашего веб-приложения и будут переопределять интерфейс этого класса.

Рассмотрим подробнее некоторые выражения представленного листинга.

template\_name = **'template.html'**

Шаблон, рендеринг которого выполняет наш класс-контроллер. Сейчас в данном атрибуте указано имя-заглушка, которое потом будет переопределено.

**def** get\_context\_data(self):

**return** {}

Метод, возвращающий словарь-контекст шаблона. Данный метод мы также будем переопределять в классах-потомках.

**def** get\_template(self):

**return** self.template\_name

Метод, возвращающий имя HTML-шаблона.

**def** render\_template\_with\_context(self):

template\_name = self.get\_template()  
 context = self.get\_context\_data()  
 **return '200 OK'**, render(template\_name, \*\*context)

Метод, обеспечивающий непосредственный рендеринг шаблона **template\_name** с передачей в него контекста **context**.

**def** \_\_call\_\_(self, request):

**return** self.render\_template\_with\_context()

Вспомним, что контроллер выполняет обработку некоторого запроса. Если контроллер реализовывать в виде объекта класса, то у этого класса нужно перегрузить метод **\_\_call\_\_()**. Т.е. объект этого класса будет вызываться как функция, принимать запрос request и выполнять его обработку путем вызова уже знакомого метода **render\_template\_with\_context()**.

Но **TemplateView** – базовый класс для других CBV. Он нужен, чтобы в других CBV мы каждый раз не прописывали всю описанную выше логику и избежали бы дублирования.

# применяем первый собственный встроенный класс-обработчик

Пришло время создать CBV-классы, которые мы сделаем потомками представленного ранее класса **TemplateView**.

**Листинг 2. step\_6/components/cbv.py**

|  |
| --- |
| *# Класс-контроллер для отображения списков записей* **class** ListView(TemplateView):  queryset = []  template\_name = **'list.html'** context\_object\_name = **'objects\_list'   def** get\_queryset(self):  print(self.queryset)  **return** self.queryset   **def** get\_context\_object\_name(self):  **return** self.context\_object\_name   **def** get\_context\_data(self):  queryset = self.get\_queryset()  context\_object\_name = self.get\_context\_object\_name()  context = {context\_object\_name: queryset}  **return** context |

Начнем с класса **ListView**. Значение атрибута **template\_name** здесь также будет в качестве заглушки. А реальное значение атрибута мы будем переопределять уже в классах-контроллерах нашего веб-приложения.

**def** get\_context\_data(self):  
 queryset = self.get\_queryset()  
 context\_object\_name = self.get\_context\_object\_name()  
 context = {context\_object\_name: queryset}  
 **return** context

Класс **ListView** должен обеспечивать визуализацию в шаблоне набора записей. Этот набор должен передаваться как контекст шаблона. Поэтому переопределим метод **get\_context\_data()**.

В контекст мы будем передавать переменную **'objects\_list'**, а ее значением будет массив записей. Кстати, в Django при работе с CBV применяется такая же переменная контекста шаблона. Ну а сам массив записей мы будем формировать уже когда выполним наследование интерфейса класса **ListView** в нашем веб-приложении. Это будет выглядеть, например, так:

**Листинг 3. step\_6/views.py**

|  |
| --- |
| **from** components.cbv **import** ListView  @AppRoute(routes=routes, url=**'/student-list/'**) **class** StudentListView(ListView):  queryset = site.students  template\_name = **'student\_list.html'** |

Этот класс-контроллер обеспечивает вывод списка студентов, добавившихся на наши курсы. В этом классе мы переопределяем значение атрибута **template\_name**, т. е. указываем имя шаблона, рендеринг которого выполняется. Переменная **queryset** ссылается на список записей-студентов. С атрибутом **students** мы уже знакомы. Он находится в нашем центральном классе-движке **Engine** файла **models.py**. Атрибут **queryset** определяется в классе **ListView**, но там значение-заглушка. А в классе **StudentListView** мы уже переопределяем значение этого атрибута.

# Реализуем второй собственный встроенный класс-обработчик

Опишем следующий класс CBV, интерфейс которого унаследуем от уж знакомого класса **TemplateView**. Это знакомый Django-разработчикам класс **CreateView**.

**Листинг 4. step\_6/components/cbv.py**

|  |
| --- |
| *# Класс-контроллер для создания записи* **class** CreateView(TemplateView):  template\_name = **'create.html'** @staticmethod  **def** get\_request\_data(request):  **return** request[**'data'**]   **def** create\_obj(self, data):  **pass   def** \_\_call\_\_(self, request):  **if** request[**'method'**] == **'POST'**:  *# метод пост* data = self.get\_request\_data(request)  self.create\_obj(data)   **return** self.render\_template\_with\_context()  **else**:  **return** super().\_\_call\_\_(request) |

Представленный класс применяется для создания записей, например, категорий, товаров, студентов.

**def** get\_request\_data(request):  
 **return** request[**'data'**]

Каждая запись описывает некоторую сущность, параметры которой мы будем получать из запроса с помощью метода **get\_request\_data()**.

**def** create\_obj(self, data):  
 **pass**

Опираясь на полученные данные, мы будем создавать объект нужной записи через метод **create\_obj()**.

Пока этот метод выполнен в качестве заглушки по аналогии с предыдущими листингами кода. Мы выполним переопределение метода в классах-потомках.

Конечно нам не обойтись без перегрузки метода **\_\_call\_\_()**. В нем мы при выполнении POST-запроса будем создавать запись вызовом метода **create\_obj()** и выполнять рендеринг шаблона. Функция для рендеринга уже определена нами ранее в классе **TemplateView**.

# Применяем второй собственный встроенный класс-обработчик

Теперь займемся контроллером, например, для создания записей-студентов. Этот класс-контроллер будет наследовать интерфейс класса **ListView**.

**Листинг 5. step\_6/views.py**

|  |
| --- |
| **from** components.cbv **import** CreateView  @AppRoute(routes=routes, url=**'/create-student/'**) **class** StudentCreateView(CreateView):  template\_name = **'create\_student.html'   def** create\_obj(self, data: dict):  name = data[**'name'**]  name = site.decode\_value(name)  new\_obj = site.create\_user(**'student'**, name)  site.students.append(new\_obj) |

В приведенном классе мы выполняем переопределение метода **create\_obj()**, т.е. по сути создаем новую запись-сущность для модели «Студент». Опять же инструкции:

new\_obj = site.create\_user(**'student'**, name)  
site.students.append(new\_obj)

Данные выражения нам уже знакомы по классу **Engine**. В рамках этих выражений созданная новая запись попадает в список записей, которые мы потом передадим в базу данных. На данный момент наш проект пока не связан с базой данных.

Также нам необходим контроллер для добавления студента на курс.

**Листинг 6. step\_6/views.py**

|  |
| --- |
| @AppRoute(routes=routes, url=**'/add-student/'**) **class** AddStudentByCourseCreateView(CreateView):  template\_name = **'add\_student.html'   def** get\_context\_data(self):  context = super().get\_context\_data()  context[**'courses'**] = site.courses  context[**'students'**] = site.students  **return** context   **def** create\_obj(self, data: dict):  course\_name = data[**'course\_name'**]  course\_name = site.decode\_value(course\_name)  course = site.get\_course(course\_name)  student\_name = data[**'student\_name'**]  student\_name = site.decode\_value(student\_name)  student = site.get\_student(student\_name)  course.add\_student(student) |

Он схож с предыдущим контроллером и также наследует интерфейс класса **CreateView**. При этом мы переопределяем метод **get\_context\_data()** с передачей в контекст шаблона списков с курсами и студентами, поскольку мы должны иметь возможность выбора на странице студента и курса, на который его необходимо добавить. При этом будут создаваться соответствующие записи-привязки студентов к курсам, поэтому мы переопределяем метод **create\_obj()**.

Итак, в рамках текущего урока мы создали файл **cbv.py** с базовыми классами-контроллерами, а также применили их для реализации контроллеров нашего веб-приложения, доработав файл **views.py**.

# Дополняем проект необходимыми шаблонами

Теперь наш проект имеет бэкенд для создания студентов, отображения их списка и добавления студента на курс. Теперь дело за фронтендом. Создадим HTML-шаблоны для создания студентов, отображения их списка и добавления студента на курс.

**Листинг 7. step\_6/templates/create\_student.html**

|  |
| --- |
| {% extends "base.html" %} {% block title %} Создание студента {% endblock %} {% block style %}  {% include "inc-style.html" %} {% endblock %} {% block menu %}  {% include "inc-menu.html" %} {% endblock %} {% block body %}  <**div**>  <**li**>  <**a href="/student-list/"**>Список студентов</**a**>  </**li**>  <**li**>  <**a href="/add-student/"**>Добавить студента на курс</**a**>  </**li**>  </**div**>  <**div**>  <**h1**>Создание студента</**h1**>  <**form method="post"**>  <**input type="text" name="name" placeholder="Имя"**>  <**button type="submit"**>Сохранить</**button**>  </**form**>  </**div**> {% endblock %} |

**Листинг 8. step\_6/templates/student\_list.html**

|  |
| --- |
| {% extends "base.html" %} {% block title %} Создание студента {% endblock %} {% block style %}  {% include "inc-style.html" %} {% endblock %} {% block menu %}  {% include "inc-menu.html" %} {% endblock %} {% block body %}  <**div**>  <**li**>  <**a href="/create-student/"**>Создать студента</**a**>  </**li**>  <**li**>  <**a href="/add-student/"**>Добавить студента на курс</**a**>  </**li**>  </**div**>  <**h1**>Список студентов</**h1**>  <**div**>   {% for item in objects\_list %}  <**li**>  {{item.name}}  {% for course in item.courses %}  {{course.name}}  {% endfor %}  </**li**>  {% endfor %}  </**div**> {% endblock %} |

**Листинг 9. step\_6/templates/add\_student.html**

|  |
| --- |
| {% extends "base.html" %} {% block title %} Создание курса {% endblock %} {% block style %}  {% include "inc-style.html" %} {% endblock %} {% block menu %}  {% include "inc-menu.html" %} {% endblock %} {% block body %}  <**div**>  <**li**>  <**a href="/student-list/"**>Список студентов</**a**>  </**li**>  <**li**>  <**a href="/create-student/"**>Создать студента</**a**>  </**li**>  </**div**>  <**h1**>Добавление студента на курс</**h1**>  <**form method="post"**>  <**select size="3" name="course\_name"**>  {% for item in courses %}  <**option value="{{item.name}}"**>{{item.name}}</**option**>  {% endfor %}  </**select**>  <**select size="3" name="student\_name"**>  {% for item in students %}  <**option value="{{item.name}}"**>{{item.name}}</**option**>  {% endfor %}  </**select**>  <**button type="submit"**>Записать</**button**>  </**form**> {% endblock %} |

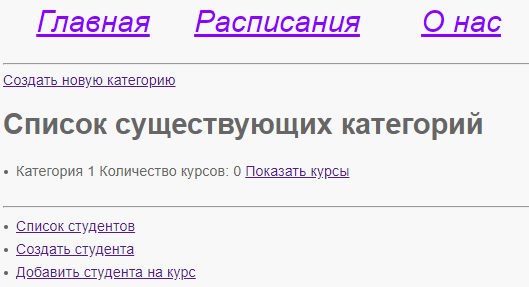
Также нам нужно доработать некоторые другие служебные классы проекта, дополнить их кодом. Интерфейс данных классов изменится, поскольку ранее класс **User** не содержал ни атрибутов, ни методов. Теперь же мы решили добавить пользователю атрибут **name**.

Также изменится интерфейс класса **Student**, добавим атрибут, содержащий список курсов, куда записан студент. В класс **Course** добавим перегрузку метода **\_\_getitem\_\_()**, чтобы реализовать возможность итерирования объекта класса **Course**, при этом мы будем обращаться к элементам атрибута **students** класса **Course**. Также добавим метод **add\_student()** для реализации возможности добавления студента на курс.

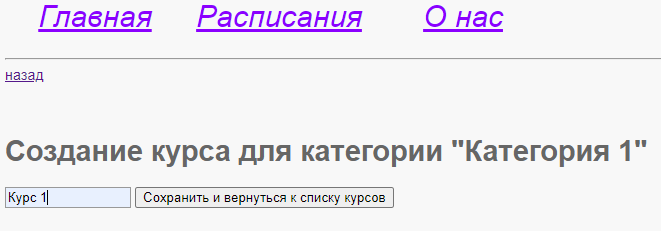
# проверяем работу проекта

Проверим работу нашего веб-приложения.

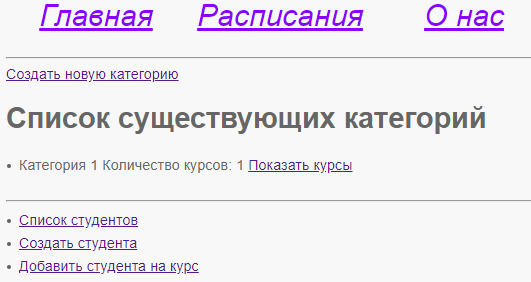
Перейдем на главную страницу и создадим категорию курсов.

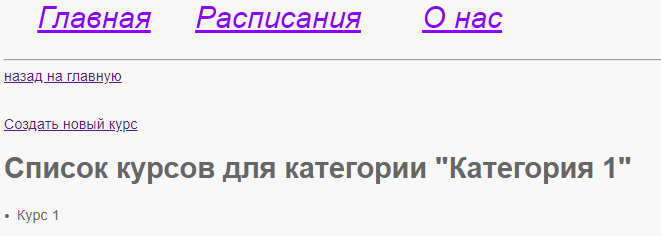


Создадим курс, относящийся к созданной категории.

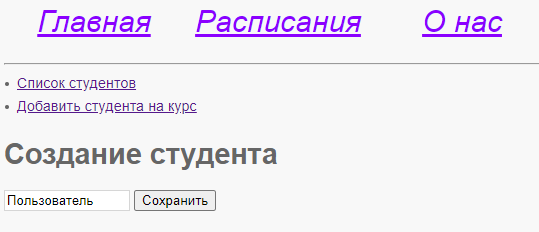


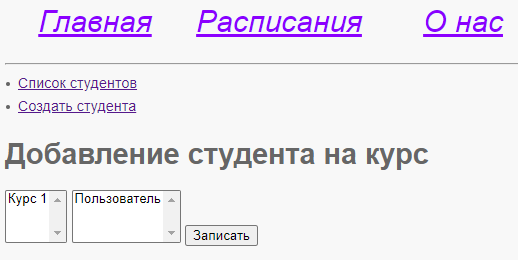
Теперь у нас есть категория и привязанный к ней курс.

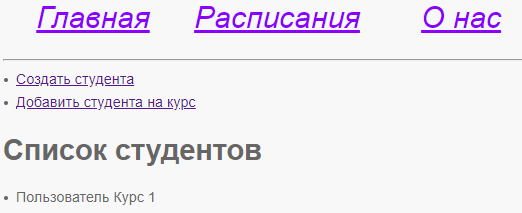




Наконец, создадим нового студента и добавим его на курс.







# Вид сверху на рукопожатие над деловыми документамиЗаключение

Теперь мы реализовали для нашего фреймворка служебные классы-аналоги классам CBV в Django. Мы сделали пока три класса: TemplateView, ListView, CreateView. Разработчики, работающие с Django без труда поймут их назначение. Интерфейс этих классов наследуют классы-контроллеры нашего веб-приложения.

## КЛЮЧЕВЫЕ ВЫВОДЫ

* Благодаря механизму наследования мы реализовали специальные служебные классы, посредством которых мы можем отображать список записей, создавать новые записи и т.д.
* При необходимости список таких встроенных классов может быть расширен, в частности, путем добавления классов для редактирования и удаления записей.

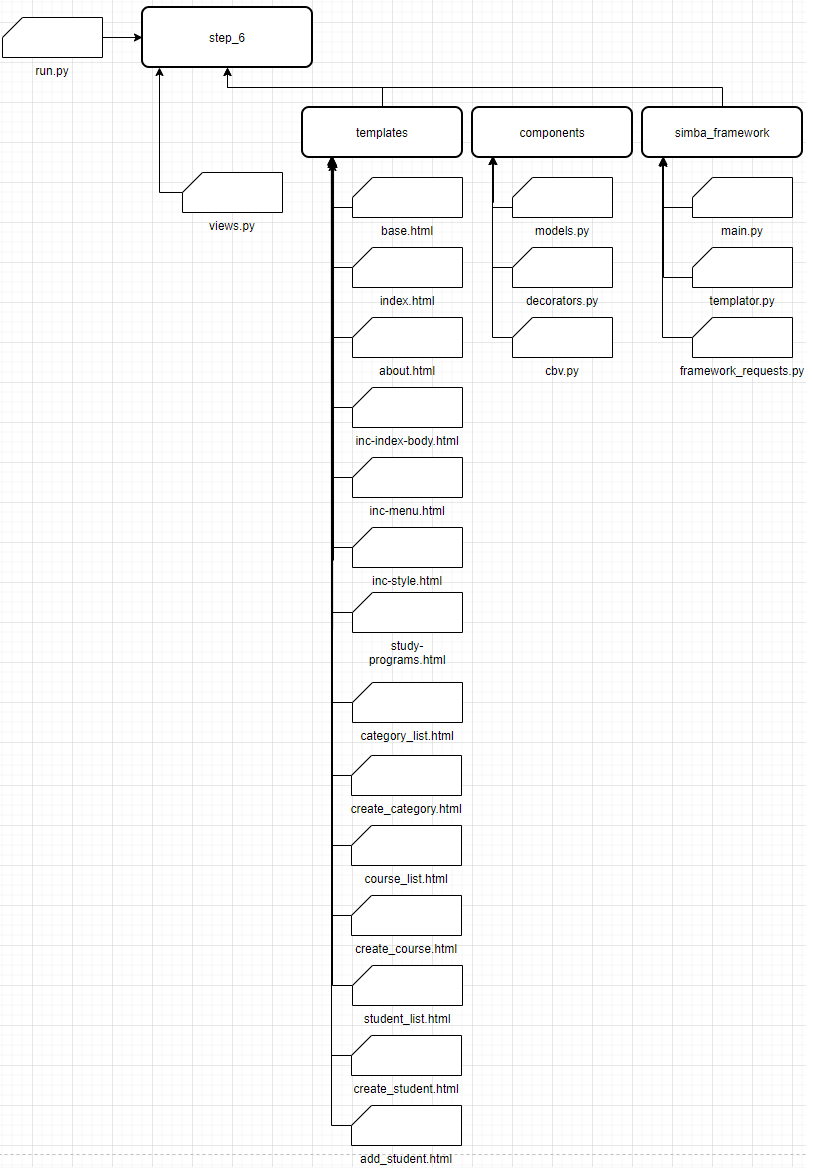
Текущая структура предложенного WSGI-фреймворка и создаваемого на его основе веб-приложения приведена на рис. 1.

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