


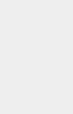
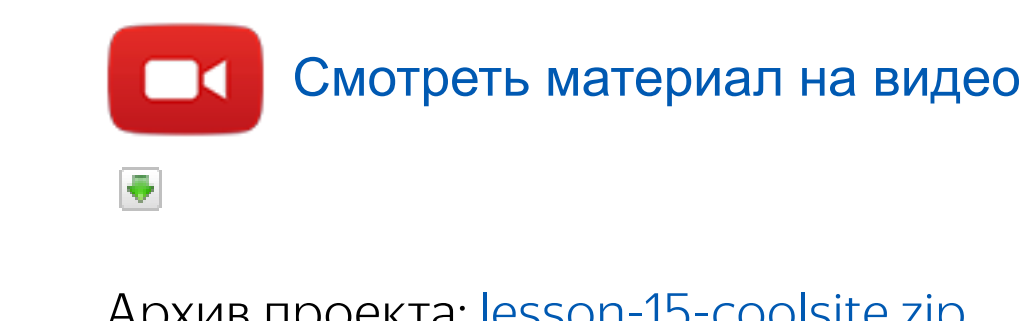


Классы представлений, регистрация, оптимизация
Классы представлений: ListView, DetailView, CreateView
Основы ORM Django за час
Mixins - убираем дублирование кода
Постраничная навигация (пагинация)
Регистрация пользователей на сайте
Делаем авторизацию пользователей на сайте
Оптимизация сайта с Django Debug Toolbar
Включаем кэширование данных
Использование капчи captcha
Тонкая настройка админ панели
Поделиться
  
Наш канал
 YouTube

Главная → Django → Классы представлений, регистрация, оптимизация

Классы представлений: ListView, DetailView, CreateView



Архив проекта: [lesson-15-coolsite.zip](#)

До сих пор в проекте мы с вами использовали функции представления или, как еще говорят, контроллеры функций. Это довольно простой подход и используется, в основном, когда нужно реализовать простую логику обработки запросов. Однако, во фреймворке Django помимо функций можно использовать и классы представлений (контроллеры классов):

CBV – Class-Based Views

Чаще всего именно их применяют на практике, так как объектно-ориентированный подход, зачастую, позволяет заметно упростить код, сделать его более понятным и читабельным.

Чтобы у вас сложилось полное представление, что такое классы представлений и как ими пользоваться, помимо этого и последующих занятий, я рекомендую также почитать русскоязычную документацию (раздел «Представления-классы»):

<https://djangobook.ru/ref/3.0/index.html>

Обычно, классы представлений создаются на основе базовых классов фреймворка Django. Полный их список можно посмотреть на этой странице:

<https://djangobook.ru/ref/9/ref/class-based-views/index.html>

Конечно, мы рассмотрим лишь некоторые из них, чтобы понять логику работы с ними. Давайте, для начала, заменим функцию представления index (файл women/views.py) на класс. Так как у нас на главной странице отображается список статей, то здесь подойдет уже готовленный для этих целей базовый класс ListView. Мы его вначале импортируем:

```
from django.views.generic import ListView
```

Затем, определим свой собственный класс представления на базе этого класса ListView, например, так:

```
class WomenHome(ListView):
    model = Women
```

а внутри определим один атрибут model, который будет ссылаться на модель данных Women, связанной с этим списком. (Так как нам на главной странице нужно вывести список женщин из этой модели). Фактически, строчка model = Women выберет все записи из таблицы women и попытается отобразить их в виде списка, используя шаблон с именем:

```
<имя приложения>/<имя модели>_list.html
```

Мы пока его прописывать не будем, а свяжем маршрут главной страницы с классом WomenHome. Переходим в файл women/urls.py и вместо строчки:

```
path('', index, name='home'),
```

запишем:

```
path('', WomenHome.as_view(), name='home'),
```

Обратите внимание, метод as_view() нужно вызвать, то есть, поставить в конце круглые скобки, а не просто передать ссылку на него. Это метод одного из базовых классов вида и служит для привязки класса представления к текущему маршруту.

Если теперь открыть главную страницу сайта:

<http://127.0.0.1:8000>

то увидим ошибку «TemplateDoesNotExist», так как фреймворк не находит шаблон по умолчанию:

women/women_list.html

Конечно, мы могли бы создать такой шаблон, но у нас уже есть свой собственный для этих целей – women/index.html и воспользуемся им. Чтобы его указать в классе представлений, используется атрибут template_name, которому присваиваем путь к нужному шаблону:

```
class WomenHome(ListView):
    model = Women
    template_name = 'women/index.html'
```

Если теперь обновить главную страницу, то увидим пустой список так, словно у нас нет ни одной статьи. Почему это произошло? Дело в том, что мы в шаблоне обращаемся к переменным со своими именами, которые определили в функции представления index. Например, post содержал список всех записей. Что же теперь нужно написать вместо post? По умолчанию, данные из модели Women, указанной в классе представлений, помещаются в коллекцию object_list и если мы ее запишем вместо post, то должно все заработать. И, действительно, обновляя главную страницу, видим список всех постов.

Как видите, чтобы получить тот же результат для главной страницы, нам потребовалось описать класс всего тремя строчками.

Если в шаблоне вместо object_list мы хотим использовать другое обозначение (имя), то в классе HomeView следует прописать атрибут context_object_name с указанием другого имени переменной:

```
context_object_name = 'posts'
```

И, далее, в шаблоне index.html снова можем писать posts. Осталось в index.html передать заголовок страницы. В нашем случае – это строка параметра title. Для передачи шаблону таких статичных данных, можно использовать специальный словарь extra_context:

```
extra_context = {'title': 'Главная страница'}
```

Но, обратите внимание, этот словарь можно использовать именно для статичных (не изменяемых) данных, такие как строки, числа. Если же мы собираемся передавать динамические данные, вроде списков, то для этого уже нужно переопределять метод get_context_data базового класса. В принципе, с помощью метода get_context_data() можно передавать и статические и динамические данные, то есть, любую информацию в шаблон. Поэтому, чаще всего, вместо extra_context используют метод get_context_data().

Интегрированная среда предлагает полный формат записи этого метода со всеми параметрами:

```
def get_context_data(self, *, object_list=None, **kwargs):
```

Я его так и оставлю, хотя, отсюда можно убрать то, что не используется. Далее, внутри метода мы первым делом должны повторить работу этого метода базового класса, используя строчку:

```
def get_context_data(self, *, object_list=None, **kwargs):
    context = super().get_context_data(**kwargs)
```

Здесь super() – это обращение к базовому классу и, далее, через точку, идет вызов аналогичного метода с передачей ему возможных именованных параметров из словаря kwargs. Сформированный базовый контекст мы сохраняем через переменную context.

В действительности, переменная context ссылается на словарь и, если нам нужно добавить какой-либо ключ, или изменить уже существующий, то достаточно записать:

```
context['title'] = 'Главная страница'
```

Здесь мы формируем параметр title, который станет доступным в шаблоне index.html. В конце метод get_context_data должен вернуть сформированный контекст:

```
def get_context_data(self, *, object_list=None, **kwargs):
    context = super().get_context_data(**kwargs)
    context['title'] = 'Главная страница'
    return context
```

Все, атрибут extra_context можно убрать и после обновления главной страницы увидим тот же результат. Добавим еще в этот метод параметр cat_selected со значением 0 для выбора нужной рубрики в шаблоне и список для главного меню:

```
context['cat_selected'] = 0
context['menu'] = menu
```

Последний штрих, который мы сделаем в классе представления WomenHome – это добавим фильтрацию статей по флагу is_published, то есть, будем отображать только опубликованные статьи. Для этого существует специальный метод get_queryset, переопределяя который, можно указать как именно выбирать записи из модели Women:

```
def get_queryset(self):
    return Women.objects.filter(is_published=True)
```

Очевидно, здесь можно воспользоваться методом filter и выбрать все записи с полем is_published в значении True. Конечно, если сейчас обновить главную страницу сайта, то никаких изменений не будет, так как у нас все записи отмечены к публикации. Но я войду в админ-панель:

<http://127.0.0.1:8000/admin/>

и в списке женщин уберу галочки у первых публикаций, сохраняем изменения и при обновлении главной страницы эти записи исчезнут из списка. Вот так, переопределяя метод get_queryset, можно задавать свой алгоритм выборки записей из модели данных.

Создаем класс представлений для категорий

Итак, мы с вами создали класс представления для главной страницы. Давайте повторим этот процесс и пропишем аналогичный класс для отдельных категорий. Делается это очень просто. Сначала объявим класс WomenCategory с тем же базовым классом ListView, указав те же самые атрибуты и метод get_queryset:

```
class WomenCategory(ListView):
    model = Women
    template_name = 'women/index.html'
    context_object_name = 'posts'

    def get_queryset(self):
        return Women.objects.filter(cat_slug=self.kwargs['cat_slug'], is_published=True)
```

Обратите внимание, как записана выборка записей. Здесь первым параметром указано имя cat_slug – это способ обращения к слугу таблицы category через объект cat модели Women. Подробнее об этом мы еще поговорим. Далее, указываем, что поле slug у категории должно быть равно параметру cat_slug, который мы берем из словаря kwargs объекта класса WomenCategory. Ключ cat_slug автоматически формируется по шаблону маршрута (файл women/urls.py), в котором мы должны вместо функции указать класс представления:

```
path('category/<slug:cat_slug>', WomenCategory.as_view(), name='category'),
```

В принципе, это все. Если теперь перейти на страницу сайта и выбирать категории, то будут отображаться статьи выбранной рубрики. Но, если указать несуществующий slug, то увидим пустую страницу, а нам бы хотелось увидеть ошибку 404 – страница не найдена. Для этого в классе WomenCategory достаточно указать атрибут:

```
allow_empty = False
```

который указывает генерировать исключение 404 если список статей пуст. Так мы сохраняем общий функционал нашего сайта.

Но вот заголовка у нас здесь никакого нет. Конечно, можно переопределить метод get_context_data и вернуть нужный title, но в учебных целях я сделаю немного иначе. В метод as_view() мы можем передавать дополнительные аргументы для классов представлений. Например, используя именованный параметр extra_context, можно передавать в класс любые статические данные, которые будут доступны в шаблоне. Мы ранее использовали это имя как атрибут класса. Здесь происходит все по аналогии: данные из extra_context будут автоматически помещены в одноименный атрибут и передаваться в шаблон. Поэтому, вот такая строчка:

```
WomenCategory.as_view(extra_context={'title': "Список по категориям"})
```

создает в шаблонах параметр title с указанным значением. Но я это все-таки уберу и сделаю через перепрыжку метода get_context_data:

```
def get_context_data(self, *, object_list=None, **kwargs):
    context = super().get_context_data(**kwargs)
    context['title'] = 'Категория - ' + str(context['posts'][0].cat)
    context['menu'] = menu
    context['cat_selected'] = context['posts'][0].cat_id
    return context
```

Смотрите, мы здесь обращаемся к выбранному записям posts, берем первую запись и обращаемся к объекту cat, который возвращает имя категории. Теперь, в заголовке страницы будет появляться информативный заголовок с именем рубрики. Также мы определяем значение параметра cat_selected с номером выбранной категории.

Правда, здесь у нас получается дублирование кода: мы каждый раз передаем главное меню, да и параметр cat_selected тоже можно оптимизировать. Но мы это сделаем в другом занятии, когда речь пойдет и миксинах (mixins).

Использование класса DetailView

Следующим шагом мы создадим еще один класс для отображения отдельных постов. Для этого хорошо подходит базовый класс DetailView. Давайте объявим класс ShowPost и унаследуем его от DetailView:

```
class ShowPost(DetailView):
    model = Women
    template_name = 'women/post.html'
```

Мы здесь сразу указали два атрибута: model – для модели; template_name – для используемого шаблона.

Далее пока ничего прописывать не будем, а перейдем к списку маршрутов (women/urls.py) и вместо строки:

```
path('post/<slug:post_slug>', show_post, name='post'),
```

запишем:

```
path('post/<slug:post_slug>', ShowPost.as_view(), name='post'),
```

Казалось бы, мы прописали базовый функционал. Но, при попытке просмотра какого-либо поста, возникает исключение «AttributeError». В чем проблема? Смотрите, вот этот класс DetailView по умолчанию пытается выбрать из указанной модели Women запись, используя атрибут pk или slug. Но у нас формируется маршрут с параметром post_slug из-за этого и возникает такая ошибка.

В самом простом случае, мы можем в шаблоне маршрута вместо post_slug записать просто slug и тогда ошибки уже не будет. Или же, в классе ShowPost прописать атрибут:

```
slug_url_kwarg = 'post_slug'
```

(Если используется идентификатор, то прописывается атрибут pk_url_kwarg). Обычно, эти атрибуты опускают и в параметрах маршрутов используют ключевые слова slug – для слага и pk – для идентификаторов.

Итак, у нас при попытке вывести статью отображается пустая страница. Возможно, мы уже догадались, это из-за использования параметра post внутри шаблона index.html. Чтобы именно такая переменная формировалась в шаблоне, мы в классе ShowPost пропишем уже знакомый нам атрибут:

```
context_object_name = 'post'
```

Все, при обновлении страницы, мы видим содержание поста. А если указать не существующий slug, то автоматически будет сгенерировано исключение 404 – страница не найдена. Как видите, все делается достаточно просто.

Осталось передать в шаблон заголовок title и пункты главного меню:

```
def get_context_data(self, *, object_list=None, **kwargs):
    context = super().get_context_data(**kwargs)
    context['title'] = context['post']
    context['menu'] = menu
    return context
```

Все, теперь у нас отображается вся нужная информация на странице.

Использование класса CreateView

В заключение этого занятия рассмотрим еще один класс CreateView для оптимизации работы с формами. Фактически, все что нам нужно сделать – это прописать в дочернем классе два атрибута:

```
class AddPage(CreateView):
    form_class = AddPostForm
    template_name = 'women/addpage.html'
```

Атрибут form_class связывает представление с классом формы AddPostForm, а template_name задает шаблон отображения формы. Если теперь связать маршрут с нашим новым представлением:

```
path('addpage/', AddPage.as_view(), name='add_page'),
```

то мы увидим поленную форму со всем необходимым функционалом. Единственное, что нужно еще добавить – это отображение главного меню и заголовка (в классе AddPage):

```
def get_context_data(self, *, object_list=None, **kwargs):
    context = super().get_context_data(**kwargs)
    context['title'] = "Добавление статьи"
    context['menu'] = menu
    return context
```

Все, работа с формой готова! Проще не придумаешь. Давайте добавим что-нибудь и при нажатии на кнопку «Добавить» мы автоматически переходим к отображению этой записи. Как это сделать? Откуда класс CreateView «узнал» куда нас следует перенаправить? Этот функционал реализуется благодаря наличию метода get_absolute_url в модели Women. Класс CreateView обращается к этому методу для получения URL-адреса добавленной статьи и перенаправляет нас к ней. Видите, как полезно следовать конвенции Django и определять стандартные методы.

Но что делать, если по каким-либо причинам в модели нет метода get_absolute_url? Для этого в классе используется AddPage нужно прописать атрибут:

```
success_url = reverse_lazy('home')
```

В этом атрибуте мы можем указать конкретную строку с URL-адресом. Но это будет хардкод. Поэтому, используется функция reverse_lazy для построения маршрутов по их именам.

Почему мы здесь вызываем именно reverse_lazy, а не просто reverse? Дело в том, что функция reverse сразу пытается построить нужный маршрут в момент создания экземпляра класса. Но это сделать невозможно, т.к. маршруты еще не были сформированы самим Django. А вот функция reverse_lazy выполняет построение маршрута только в момент, когда он понадобится. Тогда маршруты уже будут существовать и никаких проблем не возникнет.

Вообще, чтобы не запутаться в таких нюансах, можно почти всегда использовать reverse_lazy вместо reverse. На первых порах это будет безопаснее.

На этом занятии я лишь привел примеры использования различных классов представлений, чтобы вы понимали как это вообще работает и имели базовое руководство для написания начального кода. Конечно, все атрибуты, все нюансы описать достаточно сложно и это будет напоминать справочное руководство, пользы от которого больше уже не будет, т.к. такая информация быстро выветривается из головы. Главнее сейчас понять суть, принцип использования классов представлений в фреймворке Django. Ну а за более полной информацией можно обратиться к русскоязычной документации:

<https://djangobook.ru/ref/3.0/index.html>

Либо к официальной на английском языке:

<https://djangobook.ru/ref/9/ref/class-based-views/index.html>

Видео по теме

