ДЗ#3: class-based-views + пользовательские данные

Основы веб-разработки (первый семестр)

Здесь теория и рецепты для:
* Общих принципов работы class-based-views
* Работы с формами в class-based-views
* Создания и редактирования объектов с помощью generic class-based-views

Как работают class-based-views

В целом - в качестве контроллеров можно использовать либо простые функции, либо классы-наследники от django.views.generic.View.
Пока что мы использовали только последние и это, пожалуй, правильно.
Разберем основные принципы работы

Порядок работы View

Сам экземпляр контроллера создается в момент включения его в urls.py с помощью метода класса as_view.
То есть, изначально у нас есть только класс контроллера. Отдельный экземпляр появляется как только выполняется urls.py:

```
urlpatterns = [
    url(r'^$', views.QuestionList.as_view(), name='question_list'),
]
```

Вот вызов as_view как раз инициализирует экземпляр класса QuestionList и возвращает нам функцию-контроллер. Именно она потом вызывается на каждый запрос, который приходит по этой ссылке к серверу.
На самом деле эта функция-контроллер - это фактически метод dispatch у экземпляра класса View.
Вообще, честно говоря, всё сложнее, но давайте просто считать, что (в данном примере) при заходе пользователя на страницу с вопросами будет просто вызван метод dispatch у экземпляра класса QuestionList.

dispatch

При обработке запроса первым вызывается метод dispatch. Вся дальнейшая логика реализована в нем, именно этот метод обязан вернуть нам объект класса HttpResponse (то есть просто http-ответ пользователю).
Метод вызывается с request в качестве первого неименованного параметра.
Дальше следуют все параметры, которые мы забираем из урла в urls.py.
Например, если у нас вот такой urls.py:

Прямой эфир

Марина Даньшина 18 минут назад
Расписание занятий → Расписание пересдач 0

Екатерина Рогушкова 8 часов назад
Получение значков достижений на портале (ачивки) 2

Ольга Августан Вчера в 18:41
Опросы → Получение значков достижений на портале (ачивки) 2

Ольга Августан Вчера в 13:19
Общие вопросы → Завершение семестра: последние итоговые встречи 0

Ольга Августан Вчера в 13:12
Завершение семестра: итоговые встречи по дисциплинам 10

Марина Даньшина 25 Мая 2016, 14:01
Разработка на Java (первый семестр) → Забыли зонтик 0

Ksenia Sternina 25 Мая 2016, 13:38
Домашнее задание - Юзабилити-тестирование 4

Летяго Владимир 25 Мая 2016, 01:40
Разработка приложений на Android - 1 (второй семестр) → Контрольный рубеж 0

Летяго Владимир 24 Мая 2016, 22:45
Домашнее задание №3 10

Вячеслав Ишутин 24 Мая 2016, 22:31
Разработка на C++ (открытый курс) → Итоговая встреча по дисциплине 0

Вячеслав Ишутин 24 Мая 2016, 22:22
Разработка на C++ (открытый курс) → Итоговое занятие. Презентация курсовых проектов. 0

Александр Агафонов 24 Мая 2016, 19:17
Мастер-класс Ильи Стыценко онлайн «Использование OAuth 2 в приложениях на Django» 1

Марина Даньшина 23 Мая 2016, 20:59
Мероприятия → Мастер-класс Ильи Стыценко онлайн «Использование OAuth 2 в приложениях на Django» 1

Ольга Августан 23 Мая 2016, 17:56
Разработка приложений на iOS - 1 (второй

```
url(r'(?P<pk>\d+)/$', views.QuestionDetail.as_view(), name='question_detail'),
```

И пользователь заходит на ссылку /questions/15/, то будет произведен следующий вызов: QuestionDetail.dispatch(request, pk=15).

переопределение метода dispatch

Если мы в ходе обработки запроса использовать какие-то дополнительные переменные, то правильнее всего определить их в методе dispatch, в качестве аттрибутов класса, который занимается обработкой запроса.
Проще всего понять это на примере:

```
class QuestionList(ListView):
    ...
    def dispatch(self, request, *args, **kwargs):
        self.search_form = SearchForm(request.GET)
        return super(QuestionList, self).dispatch(request, *args, **kwargs)

    def get_queryset(self):
        queryset = Question.objects.all()
        queryset = queryset.filter(title=self.form.cleaned_data['search'])
        return queryset

    def get_context_data(self, **kwargs):
        context = super(QuestionList, self).get_context_data(**kwargs)
        context['search_form'] = self.search_form
        return context
```

В данном случае мы захотели прикрутить к списку объектов поиск. А значит нам нужна поисковая форма и ее данные. Вот идеальное место, чтобы инициализировать ее - именно метод dispatch. Таким образом, мы можем сделать созданную нами форму аттрибутом класса QuestionList и достать ее в любом другом методе.
В примере нам эта форма нужна в двух местах - для фильтрации списка вопросов (метод get_queryset) и для передачи в контекст шаблона (метод get_context_data). Поскольку форма уже инициализирована один раз и лежит в self.search_form - нам не нужно инициализировать ее повторно, мы просто забираем ее как аттрибут экземпляра QuestionList через self.search_form и используем там, где нам нужно. А инициализация формы в методе dispatch гарантирует нам, что форма будет инициализирована ДО вызова любых других методов.

правильное переопределение методов

В предыдущем примере мы при переопределении метода dispatch вызываем этот метод дополнительно у родителя (у класса ListView). Это нужно затем, чтобы не потерять функционал, которым dispatch занимается изначально - обработка запроса и возврат ответа. Это обычная практика в python (да и в других языках) - взяли метод класса, переопределили его, сделали в нем какие-то

семестр) → Завершение семестра 0

Ольга Августан 23 Мая 2016, 17:16
Итоговая встреча по дисциплине 2

Ольга Августан 23 Мая 2016, 17:07
Основы веб-разработки (первый семестр) → Итоговая встреча по дисциплине 0

Ольга Августан 23 Мая 2016, 17:06
Основы мобильной разработки (первый семестр) → Итоговая встреча по дисциплине 2

Ольга Августан 23 Мая 2016, 17:05
Проектирование интерфейсов мобильных приложений (второй семестр) → Итоговая встреча по дисциплине 0

Ольга Августан 23 Мая 2016, 17:02
Проектирование СУБД (второй семестр) → Итоговая встреча по дисциплине 0

Ольга Августан 23 Мая 2016, 16:59
Общие вопросы → Завершение семестра: итоговые встречи по дисциплинам 10

Весь эфир

Связь с разработчиками

Блоги

Основы веб-разработки (первый семестр) 26,04
Разработка на C++ (открытый курс) 16,97
Основы мобильной разработки (первый семестр) 14,70
Разработка на Java (первый семестр) 13,58
Общие вопросы 5,71
Разработка приложений на Android - 1 (второй семестр) 5,66
Стажировка 4,52
Проектирование интерфейсов мобильных приложений (второй семестр) 3,42
Разработка приложений на iOS - 1 (второй семестр) 2,29
Проектирование СУБД (второй семестр) 2,26

дополнительные штуки и вызвали этот метод у родителя, чтобы продолжить выполнение.
Именно так мы поступаем и с методом get_context_data (этот метод вызывается, когда наш контроллер готовит контекст для передачи в html-шаблон).
Разница между двумя этими переопределениями в том, что в случае с dispatch мы сначала меняем поведение, а потом вызываем родительский метод, чтобы продолжить выполнение. А в get_context_data мы наоборот сначала вызываем родительский метод, получаем нужный нам контекст и дописываем в него то, что нам нужно.
А вот в get_queryset мы вообще не вызываем метод родителя. На данном этапе просто не нужно.
Какие методы и как правильно переопределять у class-based-views - об этом можно почитать в официальной документации Django, там полно примеров.

Что происходит внутри вызова dispatch?

Всё просто - в dispatch проверяется метод http-запроса (request.method). Там обычно лежит строка "get" либо "post". Соответственно, вызывается метод у класса - get или post. С теми же аргументами, которые прибежали в dispatch. А дальше уже начинает работать внутренняя логика, разная для разных классов контроллеров (DetailView, ListView, и так далее).

Как забрать из ссылки больше параметров, чем предполагает View?

Например, у нас есть DetailView, показывающий отдельный Пост. Мы же хотим в ссылке использовать не только id этого поста, но и например id категории, в которой он находится.
Правильно это делать так:

```
class QuestionDetail(DetailView):
    def dispatch(self, request, category_id=None, *args, **kwargs):
        self.category = get_object_or_404(Category.objects.all(), id=category_id)
        return super(QuestionDetail, self).dispatch(request, *args, **kwargs)
```

А в urls.py:

```
url(r'(?P<category_id>\d+)/(?
P<pk>\d+)/$', views.QuestionDetail.as_view(), name='question_detail')
```

Как это работает - мы теперь забираем из урла два именованный параметра - category_id и pk. А значит, dispatch у нас вызывается так (для ссылки /questions/20/10/): QuestionDetail.dispatch(request, category_id=20, pk=10).
Мы же явно читаем из именованных аргументов метода только category_id, а вот pk у нас ложится в kwargs. Такая вот магия python - kwargs будет равен при вызове {'pk': 10}, а родительский метод dispatch будет вызван так: DetailView.dispatch(request, pk=10).
Соответственно, сам DetailView продолжит работу в штатном режиме, в то время

как мы выдрали из урла id категории, и даже вытащили объект Category из базы данных. И в любых других методах во время работы запроса можем взять текущую категорию из self.category (пофильтровать по ней посты, с которыми можно работать, передать ее в контекст и так далее).
Естественно, то же самое работает и с любыми другими class-based контроллерами.

UpdateView, CreateView

DetailView и ListView занимаются показыванием списка объектов или отдельного объекта, и уже знакомы нам.
Редактирование и создание объектов пользователем проще всего сделать с помощью UpdateView и CreateView.
Схема работы такова - при GET-запросе создается форма и рендерится html-шаблон с этой формой внутри (в переменной form, то есть мы можем сделать там как минимум {{ form }}).
При POST-запросе проверяется валидность введенных данных и если все ок - создается либо редактируется объект.
Если не ок - снова отдается html-шаблон, форма содержит ошибки валидации.
Стоит отметить, что класс UpdateView, также как и DetailView, забирает из урла параметр pk и тащит объект из базы по его айдишнику.

Методы и атрибуты:

model

Модель, с которой мы работаем, аналогично DetailView, ListView

fields

Поля объекта, с которыми разрешено работать в данном контроллере (аналогично ModelForm из предыдущего поста)

get_queryset

Аналогично DetailView и ListView, мы можем ограничить выборку объектов, с которыми работаем

get_object

Этот метод занимается предоставлением текущего объекта для редактирования

get_success_url

При успешном выполнении создания или редактирования - будет вызван этот метод. Он должен вернуть ссылку, на которую будет произведена переадресация

template_name

Имя шаблона, который рендерится при GET-запросах, аналогично DetailView, ListView

context_object_name

Имя переменной, в которой в контексте шаблона будет лежать редактируемый(создаваемый) объект, аналогично DetailView

form_class

Класс формы, которую мы хотим использовать. Обычно это наследник ModelForm по нужной нам модели

form_valid

Метод, который будет вызван в случае успешной валидации данных

object

В ходе обработки запроса текущий создаваемый(редактируемый) объект доступен через self.object

Часть из этих методов самоочевидна, либо знакома по DetailView, ListView. Про остальные - подробнее:

get_queryset

С помощью get_queryset в UpdateView мы можем ограничить объекты от редактирования не-авторами.

```
def get_queryset(self):
    return Post.objects.filter(author=self.request.user)
```

get_object

get_object вызывается когда нужно собственно получить объект, который будет создаваться или редактироваться.
По умолчанию он использует метод get_queryset (сначала получает выборку, потом с помощью нее тащит из базы объект по id)
get_object удобно переопределить, когда пользователь редактирует собственный профиль:

```
def get_object(self):
    return self.request.user
```

Оп. Редактируем собственного пользователя.

get_success_url

В случае успешного сохранения объекта мы возвращаем редирект на какую-либо страницу (например, на страницу этого самого объекта).
Метод get_success_url вызывается без параметров и должен вернуть собственно эту ссылку. Хардкодить ссылку (писать что-то вроде return "posts/"+self.object.id+"/") - нельзя и карается межгалактической лигой.
Нужно использовать хелпер resolve_url - это просто функция, которая работает полностью аналогично тегу {% url %}. На примере будет понятно:

```
from django.shortcuts import resolve_url
class QuestionCreate(CreateView):
    def get_success_url(self):
        return resolve_url('questions:question_detail', pk=self.object.pk)
```

form_class

По умолчанию в качестве формы у нас создается экземпляр класса ModelForm с соответствующими значениями в Meta.model и Meta.fields. Допустим, мы хотим где-то поменять поведение этой формы. Дописать свой clean-метод, добавить какое-нибудь поле, поменять widget для одного из полей (всё это можно переопределять в ModelForm так же как и в обычных формах). В этом случае мы создаем свою форму-наследник от ModelForm, прописываем там всё, что нужно, и определяем аттрибут form_class соответственно. Аттрибут fields у нашего classview мы тогда должны удалить

form_valid

Именно в этом методе происходит сохранение объекта. Так что, если мы хотим обновить в объекте те поля, которые не отражены в форме (например, при создании поста выставить посту автора из request.user) - мы переопределяем работу именно этого метода.
К примеру:

```
class QuestionCreate(CreateView):
    ...
    def form_valid(self, form):
        form.instance.author = self.request.user
        return super(QuestionCreate, self).form_valid(form)
```

Метод вызывается с текущей формой в качестве параметра, а у формы есть редактируемый объект в аттрибуте instance. Мы просто пишем то, что нам нужно, в этот объект, после чего вызываем родительский form_valid (он сохранит наш объект в базу и вернет редирект пользователю)

Как скрестить ежа с ужом - ListView и CreateView

Самая распространенная задачка - хотим показать пост, список комментариев, а в конце списка - формочку на добавление нового комментария.
Пока что лучший способ это сделать - реализовать создания комментария через CreateView, а функционал, которым занимается ListView реализовать самостоятельно. То есть наследуемся от CreateView, переопределяем dispatch, и там вытаскиваем из базы нужный нам пост с помощью get_objects_or_404, и

добавляем его в контекст в get_context_data. В шаблоне же всё остается практически так же как и при работе с DetailView - просто добавляем форму для создания коммента и все прочее.
Получится, что основной функционал нашего View - создание комментария, а показ поста и других каментов мы реализовали самостоятельно.
Пример навскидку (не проверял, пишу "от руки"):

```
class PostDetail(CreateView):
    model = Comment
    template_name = 'blogs/post_detail.html'
    fields = ('text', )

    def dispatch(self, request, pk=None, *args, **kwargs):
        self.post = get_object_or_404(Post, id=pk)
        return super(PostDetail, self).dispatch(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        cntext = super(PostDetail, self).get_context_data(**kwargs)
        context['post'] = self.post
        return context
```

Такая техника работает и в остальных случаях - не ленитесь писать часть функционала самостоятельно, не всегда можно положиться на дефолтную работу class-based-view

Где почитать

Начать можно отсюда https://docs.djangoproject.com/en/1.9/ref/class-based-views/ , там полно ссылок на остальные куски документации
По-русски на http://djbook.ru/

Спрашивать лучше в комментариях, чтобы остальные тоже могли прочитать ответ. Но можно и в личку.