Урок 8

Полезное: страничный вывод, шаблонные фильтры, CBV

Реализация механизма CRUD для товаров. постраничный вывод объектов. CBV: готовые контроллеры. Шаблонные фильтры.

Админка: работа с товарами

Постраничный вывод объектов

Class Based Views

- ListView
- CreateView & UpdateView
- DeleteView
- DetailView

*Собственные шаблонные фильтры

Домашнее задание

Дополнительные материалы

Используемая литература

Админка: работа с товарами

Последний урок курса. Время подводить итоги и узнать еще о нескольких полезных инструментов Django.

Начнем с реализации механизма CRUD для товаров магазина. Как и в случае с категориями, создадим форму редактирования:

adminapp/forms.py

```
...
from mainapp.models import Product
...
class ProductEditForm(forms.ModelForm):
    class Meta:
        model = Product
        fields = '__all__'

    def __init__(self, *args, **kwargs):
        super(ProductEditForm, self).__init__(*args, **kwargs)
        for field_name, field in self.fields.items():
            field.widget.attrs['class'] = 'form-control'
            field.help_text = ''
```

Пишем контроллеры.

adminapp/views.py

```
...
from adminapp.forms import ProductEditForm
...
def product_read(request, pk):
    title = 'продукт/подробнее'
    product = get_object_or_404(Product, pk=pk)
    content = {'title': title, 'object': product,}

    return render(request, 'adminapp/product_read.html', content)


def product_create(request, pk):
    title = 'продукт/создание'
    category = get_object_or_404(ProductCategory, pk=pk)

    if request.method == 'POST':
        product_form = ProductEditForm(request.POST, request.FILES)
        if product_form.is_valid():
            product_form.save()
            return HttpResponseRedirect(reverse('admin:products', args=[pk]))
    else:
        product_form = ProductEditForm(initial={'category': category})

    content = {'title': title,
```

```
            'update_form': product_form,
            'category': category
    }

    return render(request, 'adminapp/product_update.html', content)


def product_update(request, pk):
    title = 'продукт/редактирование'

    edit_product = get_object_or_404(Product, pk=pk)

    if request.method == 'POST':
        edit_form = ProductEditForm(request.POST, request.FILES,\
                                                  instance=edit_product)
        if edit_form.is_valid():
            edit_form.save()
            return HttpResponseRedirect(reverse('admin:product_update',\
                                                args=[edit_product.pk]))
    else:
        edit_form = ProductEditForm(instance=edit_product)

    content = {'title': title,
               'update_form': edit_form,
               'category': edit_product.category
    }

    return render(request, 'adminapp/product_update.html', content)


def product_delete(request, pk):
    title = 'продукт/удаление'

    product = get_object_or_404(Product, pk=pk)

    if request.method == 'POST':
        product.is_active = False
        product.save()
        return HttpResponseRedirect(reverse('admin:products',\
                                            args=[product.category.pk]))

    content = {'title': title, 'product_to_delete': product}

    return render(request, 'adminapp/product_delete.html', content)
```

Если внимательно изучить код, станет понятно, что это, по сути, повторение контроллеров для пользователей и категорий. Но есть и отличия. Здесь мы решаем новую задачу — при создании нового продукта заполняем форму начальными данными:

```
product_form = ProductEditForm(initial={'category': category})
```

Таким образом, когда пользователь нажмет кнопку «новый продукт», элемент формы, соответствующий атрибуту `'category'`, будет заполнен значением текущей категории. В именованный аргумент `initial` конструктора формы передается словарь, поэтому можно было бы заполнить остальные элементы начальными значениями.

Контроллер `product_read()` предназначен для просмотра подробной информации о продукте.

Еще одна особенность. В контроллер `product_update()` передается `pk` продукта, а нам для формирования обратной гиперссылки «к списку продуктов» необходимо в шаблон передать категорию. Но у нас же всегда есть объект категории в продукте:

```
'category': edit_product.category
```

Можно было получить объект категории прямо в шаблоне, но тогда будет ошибка при работе контроллера `product_create()`. Ведь он передает в шаблон чистую форму, объекта продукта еще нет.

Шаблоны `'product_update.html'` и `'product_delete.html'` нового кода не содержат. Вариант шаблона страницы продукта в админке:

adminapp/templates/adminapp/product_read.html

```
{% extends 'adminapp/base.html' %}
{% load staticfiles %}

{% block content %}
  <div class="product_read">
    <div class="product_name">
      Продукт<strong>{{ object.category.name|title }} /
                     {{ object.name|title }}</strong>
    </div>
    <img src="/media/{{ object.image|default:'products_images/default.jpg' }}"
         alt="{{ object.name }}">
    <div class="summary">
      <b>цена</b>
      <p>{{ object.price }} руб</p>
      <b>количество</b>
      <p>{{ object.quantity }}</p>
      <b>в каталоге</b>
      <p>{% if object.is_active %}да{% else %}нет{% endif %}</p>
      <p>
        <button>
          <a href={% url 'admin:product_update' object.pk %}>
            редактировать
          </a>
        </button>
      </p>
      <p>
        <button>
          <a href={% url 'admin:product_delete' object.pk %}>
            удалить
          </a>
        </button>
      </p>
      <p>
        <button>
          <a href={% url 'admin:products' object.category.pk %}>
            назад
          </a>
        </button>
      </p>
    </div>
    <div class="product_desc">
      <b>краткая информация</b>
      <p>{{ object.short_desc }}</p>
      <br><br>
      <b>подробная информация</b>
      <p>{{ object.description }}</p>
    </div>
  </div>
{% endblock %}
```

Не забываем прописать стили.

Постраничный вывод объектов

Мы практически завершили работу над магазином. Остались финальные штрихи. Один из них – организация постраничного вывода большого количества объектов на странице. В Django для этого есть модуль `django.core.paginator`. Рассмотрим принципы работы с ним на примере страницы каталога.

Добавляем код в контроллере.

mainapp/views.py

```
...
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
...
def products(request, pk=None, page=1):
    title = 'продукты'
    links_menu = ProductCategory.objects.filter(is_active=True)
    basket = getBasket(request.user)

    if pk:
        if pk == '0':
            category = {
                'pk': 0,
                'name': 'все'
            }
            products = Product.objects.filter(is_active=True,\
                   category__is_active=True).order_by('price')
        else:
            category = get_object_or_404(ProductCategory, pk=pk)
            products = Product.objects.filter(category__pk=pk, \
                   is_active=True, category__is_active=True).order_by('price')

        paginator = Paginator(products, 2)
        try:
            products_paginator = paginator.page(page)
        except PageNotAnInteger:
            products_paginator = paginator.page(1)
        except EmptyPage:
            products_paginator = paginator.page(paginator.num_pages)
        content = {
            'title': title,
            'links_menu': links_menu,
            'category': category,
            'products': products_paginator,
            'basket': basket,
        }

        return render(request, 'mainapp/products_list.html', content)
    ...
```

Если раньше мы передавали в шаблон переменную `products`, то теперь вместо нее `products_paginator`.

Мы передали конструктору класса `Paginator()` исходный список `products` и количество объектов на странице `'2'` и получили объект `products_paginator`, который тоже является списком, но с дополнительными атрибутами и методами. Один из них: `page()` — он позволяет получить содержимое страницы по номеру. Общее количество страниц хранится в атрибуте `num_pages`. Для предотвращения ошибок при некорректном номере страницы, переданном в адресной строке, обрабатываем исключения `PageNotAnInteger` и `EmptyPage`.

Также в коде контроллеров мы скорректировали запросы с учетом нового атрибута `is_active` в моделях продуктов и категорий продуктов — теперь в каталоге будут отображаться только активные товары и категории. **Особенность** — если категория неактивная, то товары не должны отображаться, несмотря на то, что сами они активны:

```
products = Product.objects.filter(is_active=True, category__is_active=True)
```

Необходимо скорректировать запросы в остальных контроллерах.

!Важно: в контроллере необходимо получить номер выбранной пользователем страницы. Для этого добавили третий аргумент и присвоили ему значение по умолчанию:

```
def products(request, pk=None, page=1):
```

Соответствующим образом необходимо скорректировать диспетчер URL:

mainapp/urls.py

```
from django.conf.urls import url
import mainapp.views as mainapp

urlpatterns = [
    url(r'^$', mainapp.products, name='index'),
    url(r'^category/(?P<pk>\d+)/$', mainapp.products, name='category'),
    url(r'^product/(?P<pk>\d+)/$', mainapp.product, name='product'),

    url(r'^category/(?P<pk>\d+)/page/(?P<page>\d+)/$', mainapp.products,\
                                                       name='page'),
]
```

И шаблон:

mainapp/templates/mainapp/products_list.html

```
{% extends 'mainapp/base.html' %}
{% load staticfiles %}


{% block content %}
  <div class="details">
    <div class="links clearfix">
      {% include 'mainapp/includes/inc_categories_menu.html' %}
    </div>

    <div class="products_list">
      <div class="title clearfix">
        <h2>Категория: "{{ category.name|title }}"</h2>
        <div class="paginator">
          {% if products.has_previous %}
            <a href="{% url 'products:page'
                           category.pk products.previous_page_number %}">
              <
            </a>
          {% endif %}
          <span class="current">
            страница {{ products.number }} из
            {{ products.paginator.num_pages }}
          </span>
          {% if products.has_next %}
            <a href="{% url 'products:page'
                           category.pk products.next_page_number %}">
              >
            </a>
          {% endif %}
        </div>
      </div>
      <div class="category-products clearfix">
        {% for product in products %}
          ...
```

В шаблоне использовали атрибуты объекта класса `Paginator()`: `has_previous`, `has_next`, `number`, `previous_page_number`, `next_page_number`, `paginator.num_pages`.

При формировании динамического url-адреса `'products:page'` через пробел прописываем два аргумента:

```
{% url 'products:page' category.pk products.next_page_number %}
```

Замечание: можно было поступить по-другому — сформировать адрес вида:

```
'127.0.0.1:8000/products/category/1/?page=2',
```

и потом в контроллере получить номер страницы из словаря:

```
page = request.GET['page'].
```

При этом в диспетчере URL прописывать ничего не надо было бы, и не нужно было бы добавлять третий аргумент `page` в контроллере `products()`. Но вид гиперссылки получился бы не в «стиле Django».

Class Based Views

Самое сложное оставили на финал. Вы уже заметили, что в контроллерах CRUD много повторяющегося кода. В Django есть способ все очень сильно упростить — CBV. Это, по сути, дальнейшее развитие идеи форм. Контроллер создается для модели на базе одного из классов Django. Логика реализуется в виде методов. Наша задача — понять механизм работы концепции Class Based Views на примере контроллеров админки. Кстати, те контроллеры, которые мы использовали раньше, называют иногда Function Based Views.

ListView

Для вывода списка объектов используется класс `ListView` из модуля `django.views.generic.list`. Код в контроллере:

adminapp/views.py

```
...
from django.views.generic.list import ListView
from django.utils.decorators import method_decorator


class UsersListView(ListView):
    model = ShopUser
    template_name = 'adminapp/users.html'

    @method_decorator(user_passes_test(lambda u: u.is_superuser))
    def dispatch(self, *args, **kwargs):
        return super(UsersListView,  self).dispatch(*args, **kwargs)
...
```

Чтобы контроллер заработал достаточно задать два атрибута: `model` и `template_name`. После этого корректируем диспетчер URL:

adminapp/urls.py

```
...
    url(r'^users/read/$', adminapp.UsersListView.as_view(), name='users'),
...
```

Вспомним, что контроллер — это функция. Поэтому всех классов CBV необходимо использовать статический метод `as_view()` в диспетчерах URL.

При использовании класса `ListView` мы получаем список объектов в шаблоне в переменной `object_list`. Поэтому редактируем шаблон `'adminapp/templates/adminapp/users.html'`:

{% for object in **objects** %} → {% for object in **object_list** %}

Все. Мы обеспечили тот же функционал, но прописали при этом всего две строки кода. Однако, возникают вопросы:

- как теперь ограничить доступ к админке – декораторы можно применять к функциям, но не к классам;
- как передать переменную в шаблон – например, название страницы `title`;
- как управлять запросами – например, мы хотим добавить сортировку пользователей, как это было раньше в контроллере на основе функции.

*Пока решим только первый вопрос.

Обернем отвечающий за отправку ответа в классах CBV метод `dispatch()` в специальный декоратор `@method_decorator`. А уже ему передадим знакомый нам декоратор `@user_passes_test`. Заметим, что здесь поведение `dispatch()` не изменялось – использовали метод `super()`, чтобы вернуть реализацию метода из родительского класса.

CreateView & UpdateView

Оба класса импортируются из модуля `django.views.generic.edit`. Контроллеры:

`adminapp/views.py`

```
...
from django.views.generic.edit import CreateView, UpdateView
from django.urls import reverse_lazy
...
class ProductCategoryCreateView(CreateView):
    model = ProductCategory
    template_name = 'adminapp/category_update.html'
    success_url = reverse_lazy('admin:categories')
    fields = ('__all__')

class ProductCategoryUpdateView(UpdateView):
    model = ProductCategory
    template_name = 'adminapp/category_update.html'
    success_url = reverse_lazy('admin:categories')
    fields = ('__all__')

    def get_context_data(self, **kwargs):
        context = super(ProductCategoryUpdateView, self).\
                                                get_context_data(**kwargs)
        context['title'] = 'категории/редактирование'

        return context
...
```

В атрибут класса `success_url` прописываем адрес, по которому необходимо перейти в случае успешного выполнения действий.

!Внимание: при работе с классами использование функции `reverse()` может привести к ошибке. Вместо нее используем `reverse_lazy()`. Как видно из кода, нам удалось оставить один шаблон на два контроллера. Единственная правка:

```
{{ update_form.as_p }}    →    {{ form.as_p }}
```

В классе `ProductCategoryUpdateView()` мы решили вторую задачу — передали в шаблон переменную. Для этого использовали встроенный метод `get_context_data()`. Получаем контекст как словарь:

```
super(ProductCategoryUpdateView, self).get_context_data(**kwargs)
```

Добавляем ключ `'title'` и его значение:

```
context['title'] = 'категории/редактирование'
```

Также можно было бы поработать со значениями других ключей. Можете при помощи `print()` посмотреть ключи и значения контекста шаблона.

Не забудьте прописать вызов методов `as_view()` созданных классов в диспетчере адресов вместо контроллеров-функций `category_create()` и `category_update()`!

DeleteView

Класс, предназначенный для удаления объектов, из того же модуля `django.views.generic.edit`. Код:

`adminapp/views.py`

```
...
from django.views.generic.edit import DeleteView
...
class ProductCategoryDeleteView(DeleteView):
    model = ProductCategory
    template_name = 'adminapp/category_delete.html'
    success_url = reverse_lazy('admin:categories')

    def delete(self, request, *args, **kwargs):
        self.object = self.get_object()
        self.object.is_active = False
        self.object.save()

        return HttpResponseRedirect(self.get_success_url())
...
```

Здесь мы переопределили метод `delete()`. **Обратите внимание**: не вызываем родительский метод как раньше, а просто пишем свою реализацию. После выполнения действий: возвращаем ответ — переход по адресу, который задали в атрибуте `success_url`:

```
HttpResponseRedirect(self.get_success_url())
```

В шаблоне изменяем имя переменной с `category_to_delete` на `object`. Не забываем скорректировать диспетчер адресов.

DetailView

На этом классе мы заканчиваем работу с CBV. Вершина лаконичности:

`adminapp/views.py`

```
...
from django.views.generic.detail import DetailView
...
class ProductDetailView(DetailView):
    model = Product
    template_name = 'adminapp/product_read.html'
...
```

Плюс правка в диспетчере адресов, и … контроллер готов!

Как видите, механизм CBV позволяет значительно уменьшить количество кода. Но самое главное – он позволяет работать с контроллерами при помощи ООП-подхода. Многое еще предстоит изучить, но пройденного материала уже достаточно, чтобы создавать эффективные django-приложения. В конце курса изучим еще одну тему.

*Собственные шаблонные фильтры

Выполняем обещанное в начале курса. Напишем шаблонный фильтр, который будет дописывать относительный адрес папки с медиафайлами к относительному адресу картинки, хранящемуся в модели.

Создадим в папке с приложением (например, adminapp) папку `'/templatetags/'`. Это очередная папка, в которую Django «смотрит» автоматически. Создадим в ней python-файл с любым именем, например:

adminapp/templatetags/my_tags.py

```
from django import template

register = template.Library()

URL_PREFIX = '/media/'


def media_folder_products(string):
    if not string:
        string = 'products_images/default.jpg'

    new_string = "{}{}".format(URL_PREFIX, string)

    return new_string


@register.filter(name='media_folder_users')
def media_folder_users(string):
    if not string:
        string = 'users_avatars/default.jpg'

    new_string = "{}{}".format(URL_PREFIX, string)

    return new_string


register.filter('media_folder_products', media_folder_products)
#register.filter('media_folder_users', media_folder_users)
```

Импортировали модуль `template` и создали на базе его класса `Library` объект `register`.

Написали две обычные python-функции и при помощи метода `filter()` зарегистрировали их в библиотеке фильтров:

```
register.filter('media_folder_products', media_folder_products)
```

Первый аргумент — имя, под которым фильтр будет доступен в шаблонах, второй — имя python-функции.

Можно было поступить проще — применить декоратор:

```
@register.filter(name='media_folder_users')
```

Логика самих функций простая: мы добавляем к аргументу `string` имя папки с медиафайлами, заданное в константе `URL_PREFIX`. Для вывода различных картинок по умолчанию в списке пользователей и продуктов вместо одного шаблонного фильтра сделали два. Если такой задачи нет — можно обойтись одним.

Как это использовать? Необходимо загрузить фильтры из файла в шаблон:

```
{% load my_tags %}
```

Мы уже привыкли так загружать `staticfiles`. Очевидно, что после тега `load` необходимо прописать имя файла, где заданы фильтры, только без расширения `'.py'`.

Теперь адреса картинок продуктов можно записывать в виде:

```
{{ object.image|media_folder_products }}
```

Аватарки пользователей:

```
{{ object.avatar|media_folder_users }}
```

Что это нам дает? Динамику. Мы можем изменить константу `URL_PREFIX`, и все адреса автоматически перепишутся.

Приятная новость: можно использовать фильтры в любом приложении проекта! Обязательно попробуйте в шаблонах корзины или главного приложения (mainapp).

Все. На этом наш курс завершен. Если что-то не получилось сразу – ничего страшного. Фреймворки требуют времени на освоение. Вернитесь к этим темам через некоторое время, и все покажется простым и понятным.

Спасибо за работу!

Домашнее задание

1. Реализовать работу с товарами в админке.
2. Организовать постраничный вывод в каталоге и админке.
3. Перевести как можно больше контроллеров в проекте на CBV (по крайней мере, по одному для каждого из рассмотренных классов).
4. Написать свои шаблонные фильтры и применить их.
5. *Перевести админку на AJAX.

Обязательно выполните эти задания. Это необходимо для следующего уровня!

Дополнительные материалы

Все то, о чем сказано в методичке, но подробнее:

1. Введение в CBV
2. Метод `dispatch`
3. `CreateView`
4. `UpdateView`
5. `DeleteView`
6. `DetailView`
7. Собственные шаблонные фильтры Django

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Django Book(rus)