Урок 8

Полезное: постраничный 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

Практическое задание

Дополнительные материалы

Админка: работа с товарами

Время подвести итоги и узнать еще о нескольких полезных инструментах 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().__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 — добавляя новый продукт, заполняем форму начальными данными:

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

Когда пользователь нажмет кнопку «Новый продукт», элемент формы, соответствующий атрибуту **'category'**, заполнится значением текущей категории. В именованный аргумент **initial** конструктора формы передается словарь, поэтому можно 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 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()**. Он передает в шаблон чистую форму, где 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 — организовать постраничный вывод 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 is not None:
        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**.

Передали конструктору класса **Paginator()** исходный список **products** и количество объектов на странице — '**2**'. Получили объект **products_paginator**, который тоже является списком, но с дополнительными атрибутами и методами. Один из них — **page()** — 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 не активна, то товары не должны отображаться, даже если 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 добавим третий аргумент и присвоим ему значение по умолчанию:

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

Соответствующим образом надо скорректировать диспетчер URL:

mainapp/urls.py

```
from django.urls import path

import mainapp.views as mainapp

app_name = 'mainapp'

urlpatterns = [
    path('', mainapp.products, name='index'),
    path('category/<int:pk>/', mainapp.products, name='category'),
    path('category/<int:pk>/page/<int:page>/', mainapp.products, name='page'),
    path('product/<int:pk>/', mainapp.product, name='product'),
]
```

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 иначе — 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 потребовалось 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 капитально упростить — **CBV**. Это 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().dispatch(*args, **kwargs)
...
```

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

adminapp/urls.py

```
...
path('users/read/', adminapp.UsersListView.as_view(), name='users'),
...
```

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

При использовании класса **ListView** 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 %}
```

Обеспечили тот же функционал, но прописали при этом всего две строки кода. Но 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, чтобы добавить сортировку пользователей, как это было в контроллере на основе функции.

* Пока решим только первый вопрос.

Обернем метод dispatch(), отвечающий за отправку ответа в классах CBV, в специальный декоратор @method_decorator. А уже ему передадим декоратор **@user_passes_test**. 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().get_context_data(**kwargs)
        context['title'] = 'категории/редактирование'

        return context
...
```

В атрибут класса **success_url** прописываем адрес, по которому необходимо перейти при успешном выполнении .

Внимание: если при работе с классами использовать функцию **reverse()**, может возникнуть ошибка. Вместо этого используем **reverse_lazy()**. Как видно из кода, удалось оставить один шаблон на два контроллера. Единственная правка:

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

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

```
super().get_context_data(**kwargs)
```

Добавляем ключ **'title'** и его значение:

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

Можно еще поработать со значениями других ключей. Можете при помощи **print()** просмотреть 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. Верх 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 сократить код. Но самое главное — дает работать с контроллерами при помощи ООП-подхода. Многое еще предстоит изучить, но пройденного материала уже достаточно, чтобы разрабатывать эффективные django-приложения.

* Собственные шаблонные фильтры

Выполняем то, что обещали 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
from django.conf import settings

register = template.Library()

def media_folder_products(string):
    """
    Автоматически добавляет относительный URL-путь к медиафайлам продуктов
    products_images/product1.jpg --> /media/products_images/product1.jpg
    """
    if not string:
        string = 'products_images/default.jpg'

    return f'{settings.MEDIA_URL}{string}'


@register.filter(name='media_folder_users')
def media_folder_users(string):
    """
    Автоматически добавляет относительный URL-путь к медиафайлам пользователей
    users_avatars/user1.jpg --> /media/users_avatars/user1.jpg
    """
    if not string:
        string = 'users_avatars/default.jpg'

    return f'{settings.MEDIA_URL}{string}'

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

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: добавляем к аргументу **string** имя папки с медиафайлами, заданное в константе **URL_PREFIX**. Для вывода 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 }}
```

Это дает динамику: 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**).

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!

Практическое 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 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.