Полезное: постраничный вывод, шаблонные фильтры, CBV

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

[Админка: работа с товарами](#_ixiuxtoe4gqy)

[Постраничный вывод объектов](#_3znysh7)

[Class Based Views](#_p9mp2xytwqsi)

[ListView](#_tyjcwt)

[CreateView & UpdateView](#_vacsjuju4bfl)

[DeleteView](#_1t3h5sf)

[DetailView](#_4d34og8)

[\* Собственные шаблонные фильтры](#_2s8eyo1)

[Практическое задание](#_17dp8vu)

[Дополнительные материалы](#_3rdcrjn)

# 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()](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View.dispatch), отвечающий за отправку ответа в классах CBV, в специальный декоратор [@method\_decorator](https://docs.djangoproject.com/en/2.1/_modules/django/utils/decorators/). 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:

|  |
| --- |
| 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) |

**adminapp/templatetags/my\_tags.py**

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](https://docs.djangoproject.com/en/2.1/topics/class-based-views/intro/).
2. [Метод dispatch](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View.dispatch).
3. [CreateView](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-editing/#django.views.generic.edit.CreateView).
4. [UpdateView](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-editing/#django.views.generic.edit.UpdateView).
5. [DeleteView](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-editing/#django.views.generic.edit.DeleteView).
6. [DetailView](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-display/).
7. [Собственные шаблонные фильтры Django](https://docs.djangoproject.com/en/2.1/howto/custom-template-tags/).