Работа с заказом пользователя: CBV, Django formsets

Создаем приложение ordersapp. Работаем с наборами форм Django formsets, используя CBV.

[Задачи](#_e4by581dlzha)

[Личный кабинет пользователя: создаем выпадающее меню](#_ky2qxrb0dej3)

[Работа с заказом: приложение «ordersapp»](#_3znysh7)

[Модели](#_7pxx2z5d9uno)

[Формы](#_2et92p0)

[Диспетчер URL приложения](#_w2mv5jrlq53y)

[Контроллер для просмотра списка заказов](#_qorl4tdhs92t)

[Контроллер для создания заказа](#_j5f8j6xb44dd)

[Контроллер для редактирования заказа](#_f5dkohxwzz1y)

[Контроллер для удаления заказа](#_7sjirja5qwck)

[Контроллер для детализированного просмотра заказа](#_ypx5sphur0ee)

[Контроллер изменения статуса заказа](#_es8wb54gjmmp)

[Практическое задание](#_1mgt3baami13)

[Дополнительные материалы](#_3j2qqm3)

[Используемая литература](#_1y810tw)

# 

# Задачи

На прошлых уроках мы много работали с аутентификацией пользователя. На сегодняшнем уроке будем работать с заказом пользователя и реализуем следующий функционал:

* выпадающее меню со ссылками на профиль и список заказов в личном кабинете пользователя;
* формирование заказа на основе содержимого корзины пользователя;
* редактирование заказа;
* детализированный просмотр заказа;
* работа со статусом заказа.

Основная задача - максимально использовать Django CBV.

# Личный кабинет пользователя: создаем выпадающее меню

Изменим код ссылки на личный кабинет пользователя в шаблоне меню приложения «mainapp»:

geekshop/mainapp/templates/mainapp/includes/inc\_menu.html

|  |
| --- |
| ... <li>  <a **href**="{% url 'contact' %}" **class**="{% if request.resolver\_match.url\_name == 'contact' %}active{% endif %}">  контакты  </a> </li> {% if user.is\_authenticated %}  <li>  <div **class**="dropdown">  <a **class**="dropdown-toggle" **href**="" **data-toggle**="dropdown">  {{ user.first\_name|default:'Пользователь' }}  <span **class**="caret"></span>  </a>  <ul **class**="dropdown-menu">  <li>  <a **href**="{% url 'auth:edit' %}">  профиль  </a>  </li>  <li>  <a **href**="{% url 'ordersapp:orders\_list' %}">  заказы  </a>  </li>  </ul>  </div>  </li> {% endif %} {% if user.is\_superuser %}  <li>  <a **href**="{% url 'admin:users' %}">админка</a>  </li> {% endif %} ... |

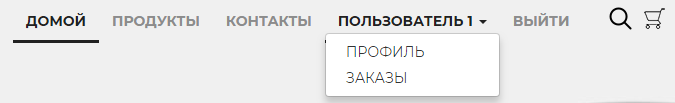
Также необходимо, чтобы в базовом шаблоне загружался Bootsrap и jQuery:

geekshop/mainapp/templates/mainapp/base.html

|  |
| --- |
| ... <head>  <meta charset="utf-8">  <title>  {% block title %}  {{ title|capfirst }}  {% endblock %}  </title>  {% block css %}  <link rel="stylesheet" type="text/css" href="{% static 'css/bootstrap.min.css' %}">  <link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">  <link rel="stylesheet" href="{% static 'fonts/font-awesome/css/font-awesome.css' %}">  {% endblock %}  {% block js %}  <script src="{% static 'js/jquery-3.2.1.min.js' %}"></script>  <script src="{% static 'js/bootstrap.min.js' %}"></script>  {% endblock %} </head> ... |

Соответствующие файлы необходимо скопировать в папку со статикой.

Если все сделано правильно, то теперь в главном меню ссылка на личный кабинет обычного пользователя должна выглядеть так:



При дальнейшем развитии проекта можно будет добавлять новые пункты в выпадающее меню.

# Работа с заказом: приложение «ordersapp»

Создаем в проекте приложение:

|  |
| --- |
| python manage.py startapp ordersapp |

*Не забываем* прописать новое приложение в настройках проекта (INSTALLED\_APPS) и создать запись в главном диспетчере URL:

|  |
| --- |
| re\_path(r'^order/', include('ordersapp.urls', namespace='order')), |

Сразу создадим базовый шаблон:

geekshop/ordersapp/templates/ordersapp/base.html

|  |
| --- |
| <!DOCTYPE html> {% load staticfiles %} <html> <head>  <meta charset="utf-8">  <title>  {% block title %}  {{ title|title }}  {% endblock %}  </title>  {% block css %}  <link rel="stylesheet" type="text/css" href="{% static 'css/bootstrap.min.css' %}">  <link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">  <link rel="stylesheet" href="{% static 'fonts/font-awesome/css/font-awesome.css' %}">  {% endblock %}  {% block js %}  <script src="{% static 'js/jquery-3.2.1.min.js' %}"></script>  <script src="{% static 'js/orders\_scripts.js' %}"></script>  {% endblock %} </head> <body>  <div class="order\_container">  {% block content %}   {% endblock %}    </div> </body> </html> |

Подключили Bootstrap, jQuery и файл, в котором будем писать JS код **«**orders\_scripts.js**»**.

## Модели

В этом приложении у нас пока будет две модели: модель заказа (Order) и модель элемента заказа (OrderItem):

geekshop/ordersapp/models.py

|  |
| --- |
| from django.db import models  from django.conf import settings from mainapp.models import Product   class Order(models.Model):  FORMING = 'FM'  SENT\_TO\_PROCEED = 'STP'  PROCEEDED = 'PRD'  PAID = 'PD'  READY = 'RDY'  CANCEL = 'CNC'   ORDER\_STATUS\_CHOICES = (  (FORMING, 'формируется'),  (SENT\_TO\_PROCEED, 'отправлен в обработку'),  (PAID, 'оплачен'),  (PROCEEDED, 'обрабатывается'),  (READY, 'готов к выдаче'),  (CANCEL, 'отменен'),  )  user = models.ForeignKey(settings.AUTH\_USER\_MODEL,  on\_delete=models.CASCADE)  created = models.DateTimeField(verbose\_name='создан', auto\_now\_add=True)  updated = models.DateTimeField(verbose\_name='обновлен', auto\_now=True)  status = models.CharField(verbose\_name='статус',  max\_length=3,  choices=ORDER\_STATUS\_CHOICES,  default=FORMING)  is\_active = models.BooleanField(verbose\_name='активен', default=True)   class Meta:  ordering = ('-created',)  verbose\_name = 'заказ'  verbose\_name\_plural = 'заказы'   def \_\_str\_\_(self):  return 'Текущий заказ: {}'.format(self.id)   def get\_total\_quantity(self):  items = self.orderitems.select\_related()  return sum(list(map(lambda x: x.quantity, items)))   def get\_product\_type\_quantity(self):  items = self.orderitems.select\_related()  return len(items)   def get\_total\_cost(self):  items = self.orderitems.select\_related()  return sum(list(map(lambda x: x.quantity \* x.product.price, items)))   *# переопределяем метод, удаляющий объект*  def delete(self):  for item in self.orderitems.select\_related():  item.product.quantity += item.quantity  item.product.save()   self.is\_active = False  self.save()   class OrderItem(models.Model):  order = models.ForeignKey(Order,  related\_name="orderitems",  on\_delete=models.CASCADE)  product = models.ForeignKey(Product,  verbose\_name='продукт',  on\_delete=models.CASCADE)  quantity = models.PositiveIntegerField(verbose\_name='количество',  default=0)   def get\_product\_cost(self):  return self.product.price \* self.quantity |

Для заказа создали минимально необходимый набор атрибутов: пользователь, время создания и обновления, статус и флаг активности. Атрибут «Статус» реализовали как поле с выбором значения из списка («choices»), с которым работали на прошлом уроке. Класс «Meta» позволяет добавить дополнительные параметры модели, например:

* сортировка по умолчанию от более новых к старым заказам:

|  |
| --- |
| ordering = ('-created',) |

* имя класса в единственном числе:

|  |
| --- |
| verbose\_name = 'заказ' |

* имя класса во множественном числе:

|  |
| --- |
| verbose\_name\_plural = 'заказы' |

Имена методов класса интуитивно понятны. Если вы переопределили метод удаления заказа - помечайте сам заказ как неактивный. Также при помощи метода «select\_related()**»** находим все элементы заказа:

|  |
| --- |
| self.orderitems.select\_related() |

и корректируем остатки продуктов на складе:

|  |
| --- |
| item.product.quantity += item.quantity  item.product.save() |

Доступ к элементам заказа через атрибут «orderitems» возможен благодаря аргументу «related\_name="orderitems"» при создании внешнего ключа order в модели элемента заказа OrderItem. Видно, что эта модель очень похожа на модель корзины «Basket».

В принципе, в качестве элемента заказа можно использовать уже существующую модель корзины Basket, добавив к ней внешний ключ «order» с аргументами «blank=True» и «null=True». Тогда при оформлении заказа связь будет формироваться через этот ключ, но при этом атрибут «user» будет избыточным - ведь он уже есть в модели заказа.

Вообще, корзина может быть реализована другими способами - например, через сессии. Поэтому правильнее создать отдельную модель для элемента заказа.

После создания моделей выполняем миграции.

## Формы

После моделей создаем формы:

geekshop/ordersapp/forms.py

|  |
| --- |
| from django import forms from ordersapp.models import Order, OrderItem  class OrderForm(forms.ModelForm):  class Meta:  model = Order  exclude = ('user',)   def \_\_init\_\_(self, \*args, \*\*kwargs):  super(OrderForm, self).\_\_init\_\_(\*args, \*\*kwargs)  for field\_name, field in self.fields.items():  field.widget.attrs['class'] = 'form-control'   class OrderItemForm(forms.ModelForm):  class Meta:  model = OrderItem  exclude = ()   def \_\_init\_\_(self, \*args, \*\*kwargs):  super(OrderItemForm, self).\_\_init\_\_(\*args, \*\*kwargs)  for field\_name, field in self.fields.items():  field.widget.attrs['class'] = 'form-control' |

Здесь, вместо разрешенных полей, прописываем кортежи «exclude = ()» с полями, которые необходимо исключить из форм. *Не забывайте* ставить запятую, даже если одно значение в кортеже.

## Диспетчер URL приложения

Перед работой с контроллерами создадим адреса в диспетчере URL:

geekshop/ordersapp/urls.py

|  |
| --- |
| import ordersapp.views as ordersapp from django.urls import re\_path  app\_name="ordersapp"  urlpatterns = [  re\_path(r'^$', ordersapp.OrderList.as\_view(), name='orders\_list'),  re\_path(r'^forming/complete/(?P<pk>\d+)/$',  ordersapp.order\_forming\_complete, name='order\_forming\_complete'),  re\_path(r'^create/$', ordersapp.OrderItemsCreate.as\_view(),  name='order\_create'),  re\_path(r'^read/(?P<pk>\d+)/$', ordersapp.OrderRead.as\_view(),  name='order\_read'),  re\_path(r'^update/(?P<pk>\d+)/$', ordersapp.OrderItemsUpdate.as\_view(),  name='order\_update'),  re\_path(r'^delete/(?P<pk>\d+)/$', ordersapp.OrderDelete.as\_view(),  name='order\_delete'), ] |

Будем реализовывать классический функционал CRUD через классы CBV.

## Контроллер для просмотра списка заказов

Начнем с простого контроллера на базе класса «ListView**»** для просмотра списка заказов:

geekshop/ordersapp/views.py

|  |
| --- |
| from django.shortcuts import get\_object\_or\_404, HttpResponseRedirect from django.urls import reverse, reverse\_lazy from django.db import transaction  from django.forms import inlineformset\_factory  from django.views.generic import ListView, CreateView, UpdateView, DeleteView from django.views.generic.detail import DetailView  from basketapp.models import Basket from ordersapp.models import Order, OrderItem from ordersapp.forms import OrderItemForm   class OrderList(ListView):  model = Order   def get\_queryset(self):  return Order.objects.filter(user=self.request.user) |

Получилось очень коротко. Переопределили метод «get\_queryset()» для того, чтобы пользователь видел только свои заказы:

|  |
| --- |
| return Order.objects.filter(user=self.request.user) |

По умолчанию Django будет искать шаблон с именем вида **«**<имя класса>\_list.html**»**. Создадим его:

geekshop/ordersapp/templates/ordersapp/order\_list.html

|  |
| --- |
| {% extends 'ordersapp/base.html' %} {% load staticfiles %}   {% block content %}  <div class="h2 text-center head">  Ваши заказы,   {% if user.first\_name %}  {{ user.first\_name|title}}  {% else %}  Пользователь  {% endif %}  </div>  <table class="table orders\_list">  <thead>  <tr>  <th>ID</th>  <th>Создан</th>  <th>Обновлен</th>  <th>Статус</th>  </tr>  </thead>  <tbody>  {% for item in object\_list %}  {% if item.is\_active %}  <tr>  <td class="td1 orders\_list">{{ item.pk }}</td>  <td>{{ item.created|date:"Y-m-d H:i:s" }}</td>  <td>{{ item.updated|date:"Y-m-d H:i:s" }}</td>  <td>{{ item.get\_status\_display }}</td>  <td>  <a href="{% url 'ordersapp:order\_read' item.pk %}">  посмотреть  </a>  </td>  <td>  {% if item.status == 'FM' %}  <a href="{% url 'ordersapp:order\_update' item.pk %}">  редактировать  </a>  {% endif %}  </td>  <td>  {% if item.status == 'FM' %}  <a href="{% url 'ordersapp:order\_delete' item.pk %}">  удалить  </a>  {% endif %}  </td>  </tr>  {% endif %}  {% endfor %}  </tbody>  </table>  <button class="btn btn-default btn-round">  <a href="{% url 'main' %}">  на главную  </a>  </button>  <button class="btn btn-default btn-round">  <a href="{% url 'ordersapp:order\_create' %}">  новый  </a>  </button> {% endblock %} |

Табличная верстка сделана для примера. Можно переписать код с использованием разметки «Bootstrap**»**.

Показываем в списке только активные заказы:

|  |
| --- |
| {% if item.is\_active %} |

Для корректного отображения атрибута модели с выбором из списка значений, используем метод «get\_<имя атрибута>\_display()**»**:

|  |
| --- |
| {{ item.get\_status\_display }} |

Некоторые действия с заказом делаем доступными только на стадии формирования:

|  |
| --- |
| {% if item.status == 'FM' %} |

## Контроллер для создания заказа

Для создания объекта модели создаем класс «OrderItemsCreate» на базе класса «CreateView». Задаем обязательные атрибуты:

|  |
| --- |
| model = Order fields = [] success\_url = reverse\_lazy('ordersapp:orders\_list') |

Список полей пустой, так как в соответствии с моделью все они, кроме пользователя «user», создаются автоматически. В поле «user» пропишем текущего пользователя позже, перед сохранением формы.

Дальше необходимо обеспечить создание элементов заказа одновременно с самим заказом. По сути, каждый элемент заказа должен создаваться на отдельной форме, для этого нам потребуется набор форм [Django FormSets](https://docs.djangoproject.com/en/2.0/topics/forms/formsets/), связанных с родительским классом (*в нашем случае это заказ «Order», а не простой, на базе класса «OrderItem»*). В Django это класс «InlineFormSet**»** ([формы на основе моделей](https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/)). Для его создания воспользуемся методом «inlineformset\_factory()**»** из модуля «django.forms**»**:

|  |
| --- |
| OrderFormSet = inlineformset\_factory(Order,   OrderItem,   form=OrderItemForm,   extra=1) |

Первый позиционный аргумент - родительский класс, второй - класс, на основе которого будет создаваться набор форм класса, указанного в именованном аргументе «form=OrderItemForm». Аргумент «extra» позволяет задать количество новых форм в наборе. Метод «inlineformset\_factory()» возвращает нам конструктор набора форм. Как и при работе с обычными формами, мы можем использовать аргументы «initial» и «instance» для передачи начальных данных или объекта в форму. Но нужно делать это для каждой формы в наборе:

|  |
| --- |
| basket\_items = Basket.get\_items(self.request.user) if len(basket\_items):  OrderFormSet = inlineformset\_factory(Order,   OrderItem,   form=OrderItemForm,  extra=len(basket\_items))  formset = OrderFormSet()  for num, form in enumerate(formset.forms):  form.initial['product'] = basket\_items[num].product  form.initial['quantity'] = basket\_items[num].quantity  basket\_items.delete() else:  formset = OrderFormSet() |

Получаем объекты корзины пользователя. Если она не пустая - создаем набор, в котором число форм равно числу объектов в корзине: «extra=len(basket\_items)». Заполняем в каждой форме поля «product» и «quantity». Чистим корзину.

Если корзина пустая - создаем набор с одной чистой формой.

Теперь необходимо передать набор форм в шаблон. При работе с Django CBV для этого используем метод «get\_context\_data(self, \*\*kwargs)». В нем необходимо получить текущий контекст:

|  |
| --- |
| data = super(OrderItemsCreate, self).get\_context\_data(\*\*kwargs) |

добавить в него данные и вернуть:

|  |
| --- |
| data['orderitems'] = formset return data |

После того, как пользователь нажмет на форме кнопку «Сохранить», создаем набор форм заново на основе данных формы, переданных методом POST:

|  |
| --- |
| if self.request.POST:  formset = OrderFormSet(self.request.POST) |

Далее происходит валидация формы - выполняется метод «form\_valid(form)» класса «CreateView». В нем сначала получим набор форм из контекста:

|  |
| --- |
| context = self.get\_context\_data() orderitems = context['orderitems'] |

Сохранение заказа и его элементов лучше выполнять как атомарную операцию - если произойдет сбой, вообще никакие данные не сохранятся. Воспользуемся для этого методом «atomic()» модуля «django.db.transaction»:

|  |
| --- |
| with transaction.atomic():  form.instance.user = self.request.user  self.object = form.save()  if orderitems.is\_valid():  orderitems.instance = self.object  orderitems.save() |

Чтобы форма создания самого заказа прошла валидацию перед сохранением, необходимо задать обязательный для модели класса «Order» атрибут «user»:

|  |
| --- |
| form.instance.user = self.request.user |

Сохраняем форму - получаем в базу данных запись для объекта заказа. Проверяем валидность набора форм с элементами заказа и сохраняем его.

Новый заказ с товарами из корзины создан. Корзина очищена.

Полный код контроллера:

geekshop/ordersapp/views.py

|  |
| --- |
| ...  class OrderItemsCreate(CreateView):  model = Order  fields = []  success\_url = reverse\_lazy('ordersapp:orders\_list')   def get\_context\_data(self, \*\*kwargs):  data = super(OrderItemsCreate, self).get\_context\_data(\*\*kwargs)  OrderFormSet = inlineformset\_factory(Order, OrderItem, \  form=OrderItemForm, extra=1)   if self.request.POST:  formset = OrderFormSet(self.request.POST)  else:  basket\_items = Basket.get\_items(self.request.user)  if len(basket\_items):  OrderFormSet = inlineformset\_factory(Order, OrderItem, \  form=OrderItemForm, extra=len(basket\_items))  formset = OrderFormSet()  for num, form in enumerate(formset.forms):  form.initial['product'] = basket\_items[num].product  form.initial['quantity'] = basket\_items[num].quantity  basket\_items.delete()  else:  formset = OrderFormSet()   data['orderitems'] = formset  return data   def form\_valid(self, form):  context = self.get\_context\_data()  orderitems = context['orderitems']   with transaction.atomic():  form.instance.user = self.request.user  self.object = form.save()  if orderitems.is\_valid():  orderitems.instance = self.object  orderitems.save()   *# удаляем пустой заказ*  if self.object.get\_total\_cost() == 0:  self.object.delete()   return super(OrderItemsCreate, self).form\_valid(form)  ... |

Здесь мы еще добавили код удаления пустого заказа.

По умолчанию при использовании классов CreateView и UpdateView шаблон должен иметь имя вида «<имя класса>\_form.html»:

geekshop/ordersapp/templates/ordersapp/order\_form.html

|  |
| --- |
| {% extends "ordersapp/base.html" %} {% load static %}  {% block content %}  {% include 'ordersapp/includes/inc\_order\_summary.html' %}  <div class="order\_form">  <form action="" method="post">  {% csrf\_token %}  {{ form.as\_p }}  <table class="table">  {{ orderitems.management\_form }}  {% for form in orderitems.forms %}  {% if forloop.first %}  <thead>  <tr>  {% for field in form.visible\_fields %}  <th class="{% cycle 'td1' 'td2' 'td3' %} order formset\_td">  {{ field.label|capfirst }}  </th>  {% endfor %}  </tr>  </thead>  {% endif %}  <tr class="formset\_row">  {% for field in form.visible\_fields %}  <td class="{% cycle 'td1' 'td2' 'td3' %} order formset\_td">  {% if forloop.first %}  {% for hidden in form.hidden\_fields %}  {{ hidden }}  {% endfor %}  {% endif %}  {{ field.errors.as\_ul }}  {{ field }}  </td>  {% endfor %}  </tr>  {% endfor %}  </table>  <button type="submit" value="сохранить" class="btn btn-default btn-round form-control last">сохранить</button>  {% include 'ordersapp/includes/inc\_order\_actions.html' %}  </form>  </div>  {% endblock %} |

Здесь подключаем два подшаблона:

* сведения о заказе (номер, дата создания и изменения, статус, количество товаров, общая стоимость):

|  |
| --- |
| inc\_order\_summary.html |

* действия на странице с заказом (совершение покупки, удаление, возврат к списку заказов, возврат на главную страницу):

|  |
| --- |
| inc\_order\_actions.html |

Они нам понадобятся, когда будем создавать контроллер для детального просмотра заказа на основе класса «DetailView». Там и рассмотрим их подробнее.

Форма создания самого заказа, при работе с CBV, передается в шаблон под именем **«**form**»** - рендерим ее при помощи метода «.as\_p()**»**:

|  |
| --- |
| {{ form.as\_p }} |

**Обязательно** выводим служебную форму для набора форм элементов заказа:

|  |
| --- |
| {{ orderitems.management\_form }} |

В ней содержатся скрытые поля с полезной информацией (количество форм в наборе, количество форм с начальными данными и т.д.). *Посмотрите* в браузере содержимое этих полей.

Сами формы набора выводим по полям. В первой итерации цикла по формам (флаг forloop.first) формируем шапку таблицы (field.label):

|  |
| --- |
| {% if forloop.first %}  <thead>  <tr>  {% for field in form.visible\_fields %}  <th class="{% cycle 'td1' 'td2' 'td3' %} order formset\_td">  {{ field.label|capfirst }}  </th>  {% endfor %}  </tr>  </thead> {% endif %} |

Шаблонный тег «cycle**»** позволяет для каждого из столбцов таблицы присвоить свой класс. При первой итерации цикла возвращается первый элемент списка, при второй - следующий и т.д.

Для формы есть возможность дополнительно вывести:

* скрытые поля:

|  |
| --- |
| form.hidden\_fields |

* ошибки поля:

|  |
| --- |
| field.errors.as\_ul |

## Контроллер для редактирования заказа

Для редактирования объекта модели на базе класса «UpdateView**»** создаем класс «OrderItemsUpdate**».** Его код будет практически совпадать с кодом предыдущего контроллера для создания заказа. Разница в создании набора форм:

|  |
| --- |
| OrderFormSet = inlineformset\_factory(Order,  OrderItem,  form=OrderItemForm,  extra=1) if self.request.POST:  data['orderitems'] = OrderFormSet(self.request.POST, instance=self.object) else:  data['orderitems'] = OrderFormSet(instance=self.object) |

При редактировании объект заказа уже существует - передаем его в набор форм:

|  |
| --- |
| OrderFormSet(instance=self.object) |

Упрощается код сохранения формы:

|  |
| --- |
| with transaction.atomic():  self.object = form.save()  if orderitems.is\_valid():  orderitems.instance = self.object  orderitems.save() |

Убрали строку:

|  |
| --- |
| form.instance.user = self.request.user |

Шаблон мы уже создали в предыдущем шаге.

## Контроллер для удаления заказа

Создаем на базе класса «DeleteView» класс «OrderDelete». Благодаря использованию CBV, код получается простым:

geekshop/ordersapp/views.py

|  |
| --- |
| ...  class OrderDelete(DeleteView):  model = Order  success\_url = reverse\_lazy('ordersapp:orders\_list')  ... |

Шаблон подтверждения удаления должен иметь имя вида «<имя класса>\_confirm\_delete.html»:

geekshop/ordersapp/templates/ordersapp/order\_confirm\_delete.html

|  |
| --- |
| {% extends 'ordersapp/base.html' %} {% load staticfiles %}  {% block content %}  <div class="category\_delete">  <div class="h1">Уверены, что хотите удалить?</div>  {% include 'ordersapp/includes/inc\_order\_summary.html' %}  <form action="{% url 'ordersapp:order\_delete' object.pk %}" method="post">  {% csrf\_token %}  <input class="btn btn-danger" type="submit" value="удалить">  </form>  <button class="btn btn-success">  <a href="{% url 'ordersapp:orders\_list' %}">отмена</a>  </button>  </div> {% endblock %} |

В шаблоне для вывода сведений о заказе можем подключить уже знакомый подшаблон «inc\_order\_summary.html**»**:

geekshop/ordersapp/templates/ordersapp/includes/inc\_order\_summary.html

|  |
| --- |
| {% if object %}  <div class="h2">Заказ №{{ object.pk }} от {{ object.created|date:"Y-m-d H:i:s" }}</div>  <hr>  <div class="h4">заказчик: {{ user.last\_name }} {{ user.first\_name }} </div>  <div class="h4">обновлен: {{ object.updated|date:"Y-m-d H:i:s" }}</div>  <div class="h4">cтатус: {{ object.get\_status\_display }}</div>  <hr>  <div class="h4">  общее количество товаров: <span class="order\_total\_quantity">{{ object.get\_total\_quantity }}</span>  </div>  <div class="h3">  общая стоимость: <span class="order\_total\_cost">{{ object.get\_total\_cost }}</span> руб  </div> {% else %}  <div class="h2">Новый заказ</div>  <hr>  <div class="h4">заказчик: {{ user.last\_name }} {{ user.first\_name }} </div>  {% if basket.0 %}  <div class="h4">  общее количество товаров: <span class="order\_total\_quantity">{{ basket.0.get\_total\_quantity }}</span>  </div>  <div class="h3">  общая стоимость: <span class="order\_total\_cost">{{ basket.0.get\_total\_cost }}</span> руб  </div>  {% endif %} {% endif %} <hr> |

Для нового заказа формируем статистику на основе модели корзины. Для существующего - на основе модели заказа.

## Контроллер для детализированного просмотра заказа

Последний контроллер на базе CBV в этом уроке - **OrderRead**. Базовый класс - «DetailView».

geekshop/ordersapp/views.py

|  |
| --- |
| ... class OrderRead(DetailView):  model = Order   def get\_context\_data(self, \*\*kwargs):  context = super(OrderRead, self).get\_context\_data(\*\*kwargs)  context['title'] = 'заказ/просмотр'  return context ... |

Это самый простой контроллер. Для передачи в шаблон данных (заголовка страницы **«**title**»**) как обычно работаем с контекстом при помощи метода «get\_context\_data()**»**.

Шаблон по умолчанию для класса «DetailView» должен иметь имя вида **«**<имя класса>\_detail.html**»**:

geekshop/ordersapp/templates/ordersapp/order\_detail.html

|  |
| --- |
| {% extends "ordersapp/base.html" %} {% load static %} {% load my\_tags %}  {% block content %}  {% include 'ordersapp/includes/inc\_order\_summary.html' %}  <div class="basket\_list">  {% for item in object.orderitems.select\_related %}  <div class="basket\_record">  <img src="{{ item.product.image|media\_folder\_products }}"  alt="{{ item.product.short\_desc }}">  <span class="category\_name">  {{ item.product.category.name }}  </span>  <span class="product\_name">{{ item.product.name }}</span>  <span class="product\_price">  {{ item.product.price }}&nbspруб  </span>  <span class="product\_quantitiy">  x {{ item.quantity }} шт.  </span>  <span class="product\_cost">  = {{ item.get\_product\_cost }}&nbspруб  </span>  </div>  {% endfor %}  </div>  {% include 'ordersapp/includes/inc\_order\_actions.html' %}  {% endblock %} |

В шаблоне подключаем оба подшаблона, упоминавшихся ранее: «inc\_order\_summary.html**»** и «inc\_order\_actions.html**»**. Код первого уже рассмотрели, код второго:

geekshop/ordersapp/templates/ordersapp/includes/inc\_order\_actions.html

|  |
| --- |
| {% if object.status == 'FM' %}  <button class="btn btn-warning btn-round form-control last">  <a href="{% url 'ordersapp:order\_forming\_complete' object.pk %}">  совершить покупку  </a>  </button>  <button class="btn btn-default btn-round form-control last">  <a href="{% url 'ordersapp:order\_delete' object.pk %}">удалить</a>  </button> {% endif %} <button class="btn btn-info btn-round form-control last">  <a href="{% url 'ordersapp:orders\_list' %}">  к списку заказов  </a> </button> <button class="btn btn-default btn-round form-control last">  <a href="{% url 'main' %}">на главную</a> </button> |

Для оформления кнопок используем классы «**Bootstrap»** (btn-round, btn-default, btn-info, btn-warning). Совершение покупки и удаление заказа делаем доступными для статуса **«**Формируется**»**:

|  |
| --- |
| {% if object.status == 'FM' %} |

Для вывода изображения товара используем, созданный на предыдущем курсе «Django 1», шаблонный фильтр **«**media\_folder\_products». Не забываем загрузить свои фильтры:

|  |
| --- |
| {% load my\_tags %} |

Напоминаем, что у нас в проекте файл с шаблонными фильтрами расположен по адресу:

|  |
| --- |
| geekshop/adminapp/templatetags/my\_tags.py |

На этом работу с CBV в уроке мы завершаем.

## Контроллер изменения статуса заказа

Для изменения статуса заказа создадим контроллер в виде функции. Здесь не нужна мощь CBV.

geekshop/ordersapp/views.py

|  |
| --- |
| ... def order\_forming\_complete(request, pk):  order = get\_object\_or\_404(Order, pk=pk)  order.status = Order.SENT\_TO\_PROCEED  order.save()   return HttpResponseRedirect(reverse('ordersapp:orders\_list')) ... |

После перехода пользователя по ссылке **«**совершить покупку**»** будет установлен статус заказа:

|  |
| --- |
| Order.SENT\_TO\_PROCEED |

Можно было бы присвоить значение «STP», которое прописано в модели. Но в дальнейшем, при изменении этого значения в модели, могут возникнуть проблемы - необходимо будет вспомнить, где еще в проекте мы используем эту константу и внести правки. Поэтому вариант присвоения константы «Order.SENT\_TO\_PROCEED», а не ее текущего значения «STP», более надежный.

# 

# Практическое задание

1. Создать выпадающее меню для ссылки на личный кабинет пользователя в меню.
2. Создать приложение для работы с заказами пользователя.
3. Создать контроллеры CRUD для заказа на базе Django CBV.
4. Реализовать обновление статуса заказа при совершении покупки.
5. Обновить контроллеры проекта, перевести на Django CBV.
6. \*Организовать работу со статусом заказов в админке (имитация обработки заказа в магазине).

# Дополнительные материалы

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

1. [Bootstrap DropDown](https://www.w3schools.com/bootstrap/bootstrap_dropdowns.asp)
2. [Django FormSets](https://docs.djangoproject.com/en/2.0/topics/forms/formsets/)
3. [Формы на основе моделей](https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/)
4. [Фабрика для InlineFormSet](https://docs.djangoproject.com/en/2.0/ref/forms/models/#django.forms.models.inlineformset_factory)
5. [ListView](https://docs.djangoproject.com/en/2.0/ref/class-based-views/generic-display/#listview)
6. [CreateView](https://docs.djangoproject.com/en/2.0/ref/class-based-views/generic-editing/#createview)
7. [UpdateView](https://docs.djangoproject.com/en/2.0/ref/class-based-views/generic-editing/#updateview)
8. [DeleteView](https://docs.djangoproject.com/en/2.0/ref/class-based-views/generic-editing/#deleteview)
9. [DetailView](https://docs.djangoproject.com/en/2.0/ref/class-based-views/generic-display/#detailview)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация](https://docs.djangoproject.com/en/2.0/)