Работа с заказом пользователя: обновляем остатки товара, добавляем код jQuery

Корректируем количество товара при работе с корзиной и заказом. Обновляем статистику заказа при его редактировании. Работаем с набором форм при помощи jQuery и django-dynamic-formset.

[Задачи](#_e4by581dlzha)

[Работа с остатками товара](#_ky2qxrb0dej3)

[Первый способ: переопределение методов моделей](#_2k5pcbgx2y73)

[Второй способ: работа с сигналами](#_qkfwac87wlmm)

[Обновление статистики заказа при его редактировании с помощью jQuery](#_262xykl1b6ui)

[Вывод дополнительной информации о цене продукта на форме заказа](#_ij2d0lica0g)

[Работа с информацией на форме при помощи jQuery](#_eizw2xy81ayf)

[Добавление новых форм к набору при помощи django-dynamic-formset](#_9h7cnuxllajd)

[Практическое задание](#_z337ya)

[Дополнительные материалы](#_3j2qqm3)

[Используемая литература](#_1y810tw)

# 

# Задачи

На прошлом уроке мы создали приложение для работы с заказами. Однако, остался нереализованным некоторый функционал:

* при манипуляциях с корзиной и заказом количество товара в модели «Product» не изменяется;
* не изменяется информация о стоимости и количестве продуктов в заказе при его редактировании;
* нет возможности добавить более одного продукта к заказу.

# Работа с остатками товара

Каждый раз при изменении количества товара или его добавлении в корзину или заказ происходит вызов метода модели «.save()». В Django мы можем решить задачу корректировки остатков товара двумя способами:

* [переопределить](https://docs.djangoproject.com/en/2.0/topics/db/models/#overriding-model-methods) методы «.save()» и «.delete()» в моделях «Basket» и «OrderItem»;
* воспользоваться механизмом [сигналов](https://docs.djangoproject.com/en/2.0/ref/signals/) (работали с ним на 2 уроке курса).

У обоих способов есть достоинства и недостатки. Второй способ лаконичнее, но сложнее с точки зрения отладки проекта.

## Первый способ: переопределение методов моделей

Работа с методом «.delete()» была рассмотрена на прошлом уроке для модели «Order». И в нем мы уже реализовали корректировку количества товаров при удалении всего заказа. Пропишем для моделей «Basket» и «OrderItem» код:

|  |
| --- |
| ... def delete(self):  self.product.quantity += self.quantity  self.product.save()  super(self.\_\_class\_\_, self).delete() |

При проверке видим, что количество товара действительно изменяется при удалении из корзины или заказа, а также при удалении всего заказа. Однако, при создании заказа из корзины обнаружим, что количество товара не изменилось, хотя элементы корзины удаляются (в коде прописано basket\_items.delete()). Это связано с тем, что при применении метода «.delete()» к QuerySet в Django, он не вызывается для каждого объекта в отдельности ([QuerySet.delete](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.delete)). Это и правильно - вместо нескольких запросов выполняется один.

В Django работа с QuerySet возможна через менеджер модели. Создадим в приложении «Basket» класс:

*geekshop/basketapp/models.py*

|  |
| --- |
| ... class BasketQuerySet(models.QuerySet):   def delete(self, \*args, \*\*kwargs):  for object in self:  object.product.quantity += object.quantity  object.product.save()  super(BasketQuerySet, self).delete(\*args, \*\*kwargs) ... |

Теперь в классе «Basket» добавим строку:

|  |
| --- |
| objects = BasketQuerySet.as\_manager() |

Можно проверять - при создании заказа остатки продуктов должны изменяться.

Конечно, можно было поступить проще - изменить код в контроллере:

*geekshop/ordersapp/views.py*

|  |
| --- |
| ... class OrderItemsCreate(CreateView):  ...  def get\_context\_data(self, \*\*kwargs):  ...  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[num].delete()**  *#basket\_items.delete()*  ... ... |

Удаляем каждый элемент корзины отдельно - срабатывает метод модели. У такого решения есть минусы: увеличивается количество запросов, в будущем, при работе с QuerySetв коде, проблема все же всплывет. Поэтому не будем использовать этот вариант.

Аналогичным образом скорректируем модель элемента заказа «OrderItem»:

*geekshop/ordersapp/models.py*

|  |
| --- |
| ... class OrderItemQuerySet(models.QuerySet):   def delete(self, \*args, \*\*kwargs):  for object in self:  object.product.quantity += object.quantity  object.product.save()  super(OrderItemQuerySet, self).delete(\*args, \*\*kwargs)   class OrderItem(models.Model):  objects = OrderItemQuerySet.as\_manager()  ... ... |

У QuerySet метода «.save()» нет (зато есть полезные в будущем методы «[bulk-create](https://docs.djangoproject.com/en/1.7/ref/models/querysets/#bulk-create)» и «[update](https://docs.djangoproject.com/en/1.7/ref/models/querysets/#update)») - переопределять нечего. Поэтому дальше работаем с методом «.save()» моделей «Basket» и «OrderItem». Тут логика будет сложнее: если мы редактируем уже существующую запись - необходимо количество оставшихся товаров изменить на разницу между прежним и новым значением в заказе (корзине):

*geekshop/basketapp/models.py*

|  |
| --- |
| ... class Basket(models.Model):  ...  *# переопределяем метод, сохранения объекта*  def save(self, \*args, \*\*kwargs):  if self.pk:  self.product.quantity -= self.quantity - \  self.\_\_class\_\_.get\_item(self.pk).quantity  else:  self.product.quantity -= self.quantity  self.product.save()  super(self.\_\_class\_\_, self).save(\*args, \*\*kwargs) ... |

Теперь все будет правильно работать - при манипуляциях с корзиной или заказом, количество товара будет корректироваться.

## Второй способ: работа с сигналами

Механизм сигналов удобен. Он широко используется в операционных системах ([сигналы в UNIX](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D1%8B_(UNIX))). В нашем случае реализация требуемого функционала через сигналы получится лаконичней. Импортируем приемник сигналов «receiver» из модуля «django.dispatch». Используем его как декоратор для функций обновления количества товаров при сохранении (product\_quantity\_update\_save) и удалении (product\_quantity\_update\_delete) объектов моделей «Basket» и «OrderItem».

geekshop/ordersapp/views.py

|  |
| --- |
| from django.dispatch import receiver from django.db.models.signals import pre\_save, pre\_delete ... @receiver(pre\_save, sender=OrderItem) @receiver(pre\_save, sender=Basket) def product\_quantity\_update\_save(sender, update\_fields, instance, \*\*kwargs):  if update\_fields is 'quantity' or 'product':  if instance.pk:  instance.product.quantity -= instance.quantity - \  sender.get\_item(instance.pk).quantity  else:  instance.product.quantity -= instance.quantity  instance.product.save()   @receiver(pre\_delete, sender=OrderItem) @receiver(pre\_delete, sender=Basket) def product\_quantity\_update\_delete(sender, instance, \*\*kwargs):  instance.product.quantity += instance.quantity  instance.product.save() |

Декоратор принимает два аргумента:

* сигнал «**pre\_save»** или «**pre\_delete»** о событии перед записью или удалением объекта ([встроенные сигналы Django](https://docs.djangoproject.com/en/2.0/ref/signals/));
* отправителя сигнала «**sender»** - класс модели, экземпляр которой будет сохранен.

Для разных сигналов набор посылаемых аргументов будет разным, мы используем следующие:

* **«sender»** - класс отправителя;
* **«update\_fields»** - имена обновляемых полей;
* **«instance»** - сам обновляемый объект

Проверяем, новый это объект или уже существующий, при помощи условия:

|  |
| --- |
| if instance.pk |

Чтобы код корректно работал, в обоих классах «Basket» и «OrderItem» должен быть реализован метод «get\_item()». Также обращаем внимание, что при использовании декоратора @receiver() у функции обязательно должен быть аргумент «\*\*kwargs».

После проверки видим, что все корректно работает. Оставим в проекте именно этот вариант.

# Обновление статистики заказа при его редактировании с помощью jQuery

В настоящее время редактирование заказа у нас происходит синхронно - при изменении количества товаров на странице, в БД реально ничего не происходит, не меняется сумма заказа и счетчик товаров в верхней части страницы. Изменения применяются при отправке формы по нажатию на кнопку «Сохранить». Если нет необходимости реализовывать асинхронное редактирование заказа (как мы это сделали для корзины на предыдущем курсе) - можно организовать обновление статистики заказа на стороне клиента средствами JS и библиотеки jQuery. Рассмотрим упрощенный вариант решения этой задачи для уже существующих элементов заказа.

Нам необходимо иметь на странице информацию о ценах продуктов. Можно поместить ее в скрытые поля формы или вывести явно. Выберем второй вариант.

## Вывод дополнительной информации о цене продукта на форме заказа

Добавляем поле «price» к форме элемента заказа:

geekshop/ordersapp/forms.py

|  |
| --- |
| class OrderItemForm(forms.ModelForm):  **price = forms.CharField(label='цена', required=False)**  ... |

Так как это поле не должно сохраняться в базу и проходить валидацию - задаем аргумент «required=False**»**. Аргумент «label**»** позволяет задать метку поля.

Поле есть. Теперь его надо заполнить:

geekshop/ordersapp/views.py

|  |
| --- |
| ... class OrderItemsCreate(CreateView):  ...   def get\_context\_data(self, \*\*kwargs):  ...   if self.request.POST:  formset = OrderFormSet(self.request.POST)  else:  basket\_items = Basket.get\_items(self.request.user)  if len(basket\_items):  ...  for num, form in enumerate(formset.forms):  form.initial['product'] = basket\_items[num].product  form.initial['quantity'] = basket\_items[num].quantity  **form.initial['price'] = basket\_items[num].product.price**  ... ...  class OrderItemsUpdate(UpdateView):  ...   def get\_context\_data(self, \*\*kwargs):  ...  if self.request.POST:  data['orderitems'] = OrderFormSet(self.request.POST,  instance=self.object)  else:  **formset = OrderFormSet(instance=self.object)  for form in formset.forms:  if form.instance.pk:  form.initial['price'] = form.instance.product.price  data['orderitems'] = formset**  return data ... |

Просто добавляем значение еще одного ключа «price» в словарь «initial**»** каждой формы набора. Причем делаем это и при создании заказа, и при его редактировании. Осталось скорректировать вывод формы в шаблоне:

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

|  |
| --- |
| ...  {% for field in form.visible\_fields %}  <td class="{% cycle 'td1' 'td2' 'td3' 'td4' %} order formset\_td">  {% if forloop.first %}  {% for hidden in form.hidden\_fields %}  {{ hidden }}  {% endfor %}  {% endif %}  {{ field.errors.as\_ul }}  **{% if field.name != 'price' %}  {{ field }}  {% else %}  {% if field.value %}  <span class="orderitems-{{forloop.parentloop.counter0}}-price">**  **{{ field.value }}**  **</span> руб  {% endif %}  {% endif %}**  </td> {% endfor %}  ... |

Будем выводить цену как обычное текстовое поле, а не как виджет:

|  |
| --- |
| {{ field.value }} |

Если для данного поля цена не задана (для нового элемента заказа) - ничего не выводим:

|  |
| --- |
| {% if field.value %} |

Добавили еще один элемент в список для именования классов столбцов таблицы (аналогично необходимо скорректировать код для шапки таблицы):

|  |
| --- |
| {% cycle 'td1' 'td2' 'td3' 'td4' %} |

Для того, чтобы считать данные о ценах товаров, необходимо задать каждому значению свой идентификатор. Это могут быть атрибуты «class» или «id**»**. Допустим наши идентификаторы должны иметь вид:

|  |
| --- |
| orderitems-<номер элемента заказа>-price |

Используем для их генерации [счетчик](https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#for) внешнего (parentloop) цикла (по формам набора), начинающийся с нуля (counter0):

|  |
| --- |
| {{forloop.parentloop.counter0}} |

Не забываем прописать стиль для класса «td4».

Итак, мы организовали вывод дополнительной информации на форме, не нарушив ее валидации.

## Работа с информацией на форме при помощи jQuery

Мы уже подключили библиотеку jQuery в базовом шаблоне при создании приложения «ordersapp» на 3 уроке. Также мы создали файл для скриптов «geekshop/static/js/orders\_scripts.js». Займемся его наполнением. Чтобы код выполнялся после загрузки DOM-модели, оборачиваем его в конструкцию:

|  |
| --- |
| window.onload = function () {} |

Сначала получим необходимые данные с формы.

Для элементов DOM модели, имеющих атрибут «value**»**, библиотека jQuery позволяет получить значение при помощи метода «.val()**»**:

|  |
| --- |
| $('<CSS селектор>').val() |

Для всех [селекторов](https://learn.javascript.ru/css-selectors), связанных с набором форм, Django добавляет префикс «<имя класса форм набора>-», в нашем случае:

|  |
| --- |
| 'orderitems-' |

Например, чтобы узнать число форм в наборе, обратимся к его служебной форме «management\_form» и ее скрытому элементу «input» с именем «orderitems-TOTAL\_FORMS»:

|  |
| --- |
| $('input[name="orderitems-TOTAL\_FORMS"]').val() |

Рекомендуем изучить содержимое этой формы в браузере (в Chrome сочетание Ctrl+Shift+I, вкладка «Elements»).

Значения обычных текстовых элементов DOM модели получаем при помощи метода «.text()**»**:

|  |
| --- |
| $('<CSS селектор>').text() |

В качестве примера приведем код для считывания со страницы стоимости заказа:

|  |
| --- |
| order\_total\_cost = parseFloat($('.order\_total\_cost').text().\ replace(',', '.')); |

Не забываем заменить запятую точкой (метод «.replace()**»**) перед преобразованием в вещественное число при помощи JS функции «parseFloat()**»**. Для преобразования в целое число будем использовать функцию «parseInt()**»**.

Далее считываем количество каждого продукта и его цену в массивы «quantity\_arr и price\_arr**»** в цикле по формам набора:

*geekshop/static/js/orders\_scripts.js*

|  |
| --- |
| ... var \_quantity, \_price, orderitem\_num, delta\_quantity, orderitem\_quantity, delta\_cost; var quantity\_arr = []; var price\_arr = [];  var TOTAL\_FORMS = parseInt($('input[name="orderitems-TOTAL\_FORMS"]').val());  var order\_total\_quantity = parseInt($('.order\_total\_quantity').text()) || 0; var order\_total\_cost = parseFloat($('.order\_total\_cost').text().\  replace(',', '.')) || 0;  for (var i=0; i < TOTAL\_FORMS; i++) {  \_quantity = parseInt($('input[name="orderitems-' + i + \  '-quantity"]').val());  \_price = parseFloat($('.orderitems-' + i + '-price').text().\  replace(',', '.'));  quantity\_arr[i] = \_quantity;  if (\_price) {  price\_arr[i] = \_price;  } else {  price\_arr[i] = 0;  } } ... |

Имена элементов «input**»** с данными модели в форме имеют вид:

«<название модели>-<номер формы в наборе>-<имя атрибута модели>»

Номера форм в наборе начинаются традиционно с нуля. Селекторы для данных с ценами товаров были сформирован нами в шаблоне в предыдущем шаге.

Если на странице данных о количестве товаров в заказе нет (например, при создании нового заказа) - вычисляем значения «order\_total\_quantity**»** и «order\_total\_cost**»**, а затем выводим их на экран при помощи jQuery метода «[.html()](http://api.jquery.com/html/)».

*geekshop/static/js/orders\_scripts.js*

|  |
| --- |
| ... if (!order\_total\_quantity) {  for (var i=0; i < TOTAL\_FORMS; i++) {  order\_total\_quantity += quantity\_arr[i];  order\_total\_cost += quantity\_arr[i] \* price\_arr[i];  }  $('.order\_total\_quantity').html(order\_total\_quantity.toString());  $('.order\_total\_cost').html(Number(order\_total\_cost.toFixed(2)).\  toString()); } ... |

Для *округления* числового значения используем JS класс-обертку «[Number()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number)»:

|  |
| --- |
| Number((<числовая переменная или операция с числами>).toFixed(2)) |

Будем обрабатывать события изменения количества или удаления товаров в заказе при помощи jQuery метода «.on()»:

|  |
| --- |
| $('.order\_form').on('click', 'input[type="number"]', function() {}) $('.order\_form').on('click', 'input[type="checkbox"]', function() {}) |

Для элементов «input» типа «number» или «checkbox» в блоке с атрибутом «class="order\_form"» по событию «click» будет выполняться соответствующий код:

geekshop/static/js/orders\_scripts.js

|  |
| --- |
| ... $('.order\_form').on('click', 'input[type="number"]', function () {  var target = event.target;  orderitem\_num = parseInt(target.name.replace('orderitems-', '').\  replace('-quantity', ''));  if (price\_arr[orderitem\_num]) {  orderitem\_quantity = parseInt(target.value);  delta\_quantity = orderitem\_quantity - quantity\_arr[orderitem\_num];  quantity\_arr[orderitem\_num] = orderitem\_quantity;  orderSummaryUpdate(price\_arr[orderitem\_num], delta\_quantity);  } });  $('.order\_form').on('click', 'input[type="checkbox"]', function () {  var target = event.target;  orderitem\_num = parseInt(target.name.replace('orderitems-', '').\  replace('-DELETE', ''));  if (target.checked) {  delta\_quantity = -quantity\_arr[orderitem\_num];  } else {  delta\_quantity = quantity\_arr[orderitem\_num];  }  orderSummaryUpdate(price\_arr[orderitem\_num], delta\_quantity); }); ... |

Получаем объект-источник события из глобального объекта события «event**»**:

|  |
| --- |
| target = event.target |

Из имени объекта (target.name) получаем номер элемента в списке форм:

|  |
| --- |
| orderitem\_num = parseInt(target.name.\  replace('orderitems-', '').replace('-quantity', '')) |

Для реализации логики определения разницы в количестве товара «delta\_quantity» используем атрибуты «.value» или «.checked» объекта «target» сохраненное в массиве предыдущее значение количества «quantity\_arr[orderitem\_num]», которое в первом обработчике не забываем обновить:

|  |
| --- |
| quantity\_arr[orderitem\_num] = orderitem\_quantity |

Значения цены товара «price\_arr[orderitem\_num]» и изменения его количества «delta\_quantity» передаем в функцию «orderSummaryUpdate» обновления статистики заказа на странице:

*geekshop/static/js/orders\_scripts.js*

|  |
| --- |
| ... function orderSummaryUpdate(orderitem\_price, delta\_quantity) {  delta\_cost = orderitem\_price \* delta\_quantity;   order\_total\_cost = Number((order\_total\_cost + delta\_cost).toFixed(2));  order\_total\_quantity = order\_total\_quantity + delta\_quantity;   $('.order\_total\_cost').html(order\_total\_cost.toString());  $('.order\_total\_quantity').html(order\_total\_quantity.toString()); }  ... |

Также необходимо внести коррективы в шаблон вывода статистики заказа:

*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>  <hr>  <div class="h4">  общее количество товаров: <span class="order\_total\_quantity"></span>  </div>  <div class="h3">  общая стоимость: <span class="order\_total\_cost"></span> руб  </div> {% endif %} <hr> |

Если сейчас полностью перезагрузить страницу редактирования заказа (Ctrl+F5) - увидим, что статистика заказа «ожила».

Главное преимущество рассмотренного решения - динамика на странице при отсутствии нагрузки на сервер. Все выполняется в браузере пользователя. Однако, наш код не будет работать для добавленного к заказу товара в имеющейся дополнительной форме на странице - для него необходимо организовать асинхронную подгрузку цены.

# Добавление новых форм к набору при помощи django-dynamic-formset

До настоящего времени число форм на странице заказа определялось в контроллере при создании набора форм. А если пользователь захочет добавить не один, а несколько новых товаров к заказу? Либо создавать сразу больше дополнительных форм, либо создавать новые формы в наборе динамически. Второе решение, разумеется, более красивое. В Django его можно реализовать при помощи одного из [пакетов](https://djangopackages.org/) [интеграции с jQuery](https://djangopackages.org/grids/g/jquery/) - «[django-dynamic-formset](https://djangopackages.org/packages/p/django-dynamic-formset/)».

Скачиваем с Github [исходник](https://github.com/elo80ka/django-dynamic-formset) и распаковываем. Из папки «src/» копируем файл «jquery.formset.js» в папку со скриптами нашего проекта «static/js/». Загружаем скрипт в базовом шаблоне сразу после библиотеки jQuery:

|  |
| --- |
| <script src="{% static 'js/jquery.formset.js' %}"></script> |

Дополним в файл со скриптами код:

*geekshop/static/js/orders\_scripts.js*

|  |
| --- |
| ... $('.formset\_row').formset({  addText: 'добавить продукт',  deleteText: 'удалить',  prefix: 'orderitems',  removed: deleteOrderItem }); ... |

Напоминаем, что он как и остальная часть скрипта должен быть *внутри*обертки:

|  |
| --- |
| window.onload = function () {} |

Можно было этот код разместить в конце шаблона редактирования заказа:

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

|  |
| --- |
| {% block content %} ... <script>  $('.formset\_row').formset({  addText: 'добавить продукт',  deleteText: 'удалить',  prefix: 'orderitems',  removed: deleteOrderItem  }); </script> ... {% endblock %} |

По сути, благодаря подключенному файлу «jquery.formset.js», получаем новый метод «.formset()» для объектов jQuery в скриптах. В него, по аналогии с «.ajax()», передаем JSON объект с параметрами:

* «**addText**» - название кнопки добавления новой формы к набору;
* «**deleteText**» - название кнопок удаления формы из набора;
* «**prefix**» - префикс имен элементов на форме (в нашем случае - имя «orderitems» класса модели формы набора);
* «**removed**» - имя пользовательской функции-обработчика удаления элемента из набора (пусть у нас это будет «deleteOrderItem»).

После обновления вид страницы редактирования заказа изменится: появится кнопка «Добавить продукт», а элементы «input» типа «checkbox» будут заменены кнопками «Удалить». Убедитесь, что все добавленные в заказ продукты сохраняются корректно.

Для корректного обновления стоимости при удалении элементов заказа, вместо второго обработчика «$('.order\_form').on('click', 'input[type="checkbox"]', function() {})», пропишем код функции «deleteOrderItem»:

*geekshop/static/js/orders\_scripts.js*

|  |
| --- |
| ... **function** deleteOrderItem(row) {  **var** target\_name= row[0].querySelector(**'input[type="number"]'**).name;  orderitem\_num = parseInt(target\_name.replace(**'orderitems-'**, **''**).\  replace(**'-quantity'**, **''**));  delta\_quantity = -quantity\_arr[orderitem\_num];  orderSummaryUpdate(price\_arr[orderitem\_num], delta\_quantity);  } ... |

Получаем массив row из одной строки, содержащей удаляемую форму набора. В этой строке находим имя элемента «input» типа «number», содержащего данные о количестве товара, и получаем из него номер формы в наборе «orderitem\_num». Дальше - обновляем данные на странице.

\*Если понадобится корректно обновлять статистику при добавлении новых элементов в заказ - можем создать еще один обработчик для изменения значения элементов «select»:

*geekshop/static/js/orders\_scripts.js*

|  |
| --- |
| ... $('.order\_form select').change(function () {  var target = event.target;  console.log(target); }); ... |

Дальше необходимо будет извлечь номер элемента в списке, при помощи AJAX получить его цену из модели и обновить ее значение на форме.

# Практическое задание

1. Организовать работу с остатками товара в проекте (попробовать оба способа).
2. Реализовать обновление статистики заказа через jQuery.
3. Расширить функционал работы с формами при помощи «django-dynamic-formset».
4. \*Реализовать асинхронное обновление цены при добавлении нового продукта в заказ.

# Дополнительные материалы

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

1. [Переопределение методов модели в Django](https://docs.djangoproject.com/en/2.0/topics/db/models/#overriding-model-methods)
2. [QuerySet.delete](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#django.db.models.query.QuerySet.delete)
3. [Сигналы в Django](https://docs.djangoproject.com/en/2.0/topics/signals/)
4. [Встроенные сигналы](https://docs.djangoproject.com/en/2.0/ref/signals/)
5. [Цикл «for» в шаблонах Django](https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#for)
6. [CSS селекторы](https://learn.javascript.ru/css-selectors)
7. [Пакеты для Django](https://djangopackages.org/)
8. [Интеграция Django и jQuery](https://djangopackages.org/grids/g/jquery/)
9. [«Django-dynamic-formset»](https://djangopackages.org/packages/p/django-dynamic-formset/)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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