Урок 4

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

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 методы «.save()» и «.delete()» в моделях «Basket» и «OrderItem»;
- воспользоваться механизмом сигналов (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). 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 QuerySet в 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» и «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). 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);
- 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 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, 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()».

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()»:

```
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">статус: {{ 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 интеграции с jQuery - «django-dynamic-formset».

Скачиваем с Github исходник 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
2. QuerySet.delete
3. Сигналы в Django
4. Встроенные сигналы
5. Цикл «for» в шаблонах Django
6. CSS селекторы
7. Пакеты для Django
8. Интеграция Django и jQuery
9. «Django-dynamic-formset»

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация