



Урок 6

Корзина + AJAX + декораторы

Работаем с корзиной. Добавляем динамику в «Горячее предложение». Делаем страницу продукта. Ограничиваем доступ к корзине. AJAX: учимся выполнять асинхронные запросы.

[Методы модели корзины](#)

[Работа с элементом «Горячее предложение»](#)

[Корзина: read и delete](#)

[Контроллеры](#)

[Шаблоны](#)

[Страница продукта](#)

[Декораторы: доступ только зарегистрированным](#)

[* AJAX: редактируем количество товаров в корзине](#)

[Практическое задание](#)

[Дополнительные материалы](#)

Методы модели корзины

На прошлом уроке мы сделали важный шаг в разработке интернет-магазина: создали корзину и вывели количество объектов в ней при помощи шаблонного фильтра. Это было не совсем правильно в рамках паттерна MVC — обработка данных должна производиться в контроллерах или моделях. Причем в Django приветствуется именно **работа с методами моделей** (толстые модели и тонкие контроллеры). Это позволяет уменьшить количество повторяющегося кода в контроллерах. Добавим три метода в модель корзины:

basketapp/models.py

```
...

@property
def product_cost(self):
    "return cost of all products this type"
    return self.product.price * self.quantity

@property
def total_quantity(self):
    "return total quantity for user"
    _items = Basket.objects.filter(user=self.user)
    _totalquantity = sum(list(map(lambda x: x.quantity, _items)))
    return _totalquantity

@property
def total_cost(self):
    "return total cost for user"
    _items = Basket.objects.filter(user=self.user)
    _totalcost = sum(list(map(lambda x: x.product_cost, _items)))
    return _totalcost

...
```

В этом коде использованы знания об ООП, полученные в курсе Python. Декоратор **property** позволяет обращаться к методу, как к свойству. В принципе, в шаблонах можно было бы вызывать и методы (как свойства). Основная идея — делать при помощи менеджера модели выборку для конкретного пользователя (**filter(user=self.user)**) и выполнять суммирование интересующих значений (количества или цены). Можно попробовать реализовать данные методы как статические.

Достаточно скорректировать подшаблон основного меню, и задача вывода стоимости корзины и полного количества товаров в ней будет решена:

```
...  
<a href="{% url 'basket:view' %}" class="basket">  
  <span>  
    {% if basket %}  
      {{ basket.0.total_cost|floatformat:0 }} руб  
      ({{ basket.0.total_quantity }} шт)  
    {% endif %}  
  </span>  
</a>
```

Особенность вывода в шаблоне в том, что надо обращаться не к списку товаров **basket**, а к конкретному товару, например: **basket.0**. Ведь методы мы делали для экземпляров модели, а не статические. Для округления стоимости товаров используем шаблонный фильтр для форматирования вывода вещественных чисел **floatformat** и передаем ему через двоеточие число десятичных разрядов **'0'**. Чтобы отображение на странице было корректным, работаем со стилями.

Работа с элементом «Горячее предложение»

У нас в верстке предусмотрен вывод «Горячего предложения» при переходе пользователя на страницу каталога. Это классический элемент любого магазина. Механизмы выбора товара на роль «горячего предложения» могут быть различными. Для примера будем просто выбирать случайный товар из каталога. В разделе «Похожие продукты» будут продукты из той же категории, что и «Горячее предложение». Редактируем контроллеры приложения **mainapp**:

```

...
import random

def get_basket(user):
    if user.is_authenticated:
        return Basket.objects.filter(user=user)
    else:
        return []

def get_hot_product():
    products = Product.objects.all()

    return random.sample(list(products), 1)[0]

def get_same_products(hot_product):
    same_products = Product.objects.filter(category=hot_product.category).\
        exclude(pk=hot_product.pk)[:3]

    return same_products

...
def products(request, pk=None):
    title = 'продукты'
    links_menu = ProductCategory.objects.all()
    basket = get_basket(request.user)

    ...

    hot_product = get_hot_product()
    same_products = get_same_products(hot_product)

    content = {
        'title': title,
        'links_menu': links_menu,
        'hot_product': hot_product,
        'same_products': same_products,
        'basket': basket,
    }

    return render(request, 'mainapp/products.html', content)

```

Чтобы избежать повторяющегося кода при обращении к корзине в каждом контроллере, мы написали функцию **get_basket(user)**. Случайный товар в функции **get_hot_product()** выбираем при помощи функции **random.sample**. Особенность: объект **QuerySet** пришлось преобразовать в обычный список. В функции **get_same_products()** пригодился метод **exclude()**, который позволил исключить само

«Горячее предложение» из списка «Похожие продукты». При помощи обычного среза [:3] ограничиваем число элементов — не более трех.

Теперь внесем изменения в шаблон:

mainapp/templates/mainapp/products.html

```
...
<div class="details-products">
  <div class="details-slider">
    <div class="slider-product">
      
    </div>
    <div class="slider-control">
      <div class="block">
        <a href="#"></a>
      </div>
      <div class="block">
        <a href="#"></a>
      </div>
      <div class="block">
        <a href="#"></a>
      </div>
    </div>
  </div>

  <div class="description">
    <h3 class="big-bold">{{ hot_product.name }}</h3>
    <h3 class="red">горячее предложение</h3>
    <p class="price">{{ hot_product.price }} <span>руб</span></p>
    <a href="{% url 'basket:add' hot_product.pk %}" class="red-button">
      заказать <i class="fa fa-chevron-right" aria-hidden="true"></i>
    </a>
    <div class="description-text">
      {{ hot_product.description }}
    </div>
  </div>
</div>
...
```

Можно проверять. При каждом переходе по ссылке «Продукты» основного меню должно появляться новое «Горячее предложение» и обновляться список «Похожие продукты».

Корзина: read и delete

Чтобы закрепить материал, реализуем просмотр корзины и удаление товаров из нее. Адреса в диспетчере URL были прописаны еще на прошлом занятии. Осталось сделать контроллеры и шаблоны.

Контроллеры

basketapp/views.py

```
...

def basket(request):
    title = 'корзина'
    basket_items = Basket.objects.filter(user=request.user).\
        order_by('product__category')

    content = {
        'title': title,
        'basket_items': basket_items,
    }

    return render(request, 'basketapp/basket.html', content)

def basket_remove(request, pk):
    basket_record = get_object_or_404(Basket, pk=pk)
    basket_record.delete()

    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Все как обычно.

Для вывода корзины получаем все объекты для текущего пользователя, упорядоченные по категориям, передаем в шаблон и рендерим.

При удалении сначала получаем объект по его **pk**, а затем вызываем метод **delete()**. Не рекомендуется реально удалять элементы из базы. Вместо этого обычно в модели создается дополнительный атрибут — флаг **active**, который устанавливают в **False**. Пока не будем усложнять.

Шаблоны

Базовый.

basketapp/templates/basketapp/base.html

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
<head>
  <meta charset="utf-8">
  <title>
    {% block title %}
      {{ title|title }}
    {% endblock %}
  </title>
  {% block css %}
    <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
    <link rel="stylesheet" href="{% static 'css/style.css' %}">
    <link rel="stylesheet" href="{% static
'fonts/font-awesome/css/font-awesome.css' %}">
  {% endblock %}
  {% block js %}
  {% endblock %}
</head>
<body>
  <div class="basket_container">
    <div class="h2 text-center head">
      Ваша корзина,
      {% if user.first_name %}
        {{ user.first_name|title }}
      {% else %}
        Пользователь
      {% endif %}
    </div>

    {% block content %}
    {% endblock %}
  </div>
</body>
</html>
```

Практически повторяет базовый шаблон **mainapp**. В принципе, можно было и не создавать отдельный базовый шаблон для **basketapp**.

Сама корзина.

basketapp/templates/basketapp/basket.html

```
{% extends 'basketapp/base.html' %}
{% load staticfiles %}

{% block content %}
<div class="basket_list">
  {% for item in basket_items %}
    <div class="basket_record">
      {{ item.product.category.name }}</span>
      <span class="product_name">{{ item.product.name }}</span>
      <span class="product_price">{{ item.product.price }}&nbsp;руб</span>
      <input type="number" name="{{ item.pk }}"
        min="0"
        value="{{ item.quantity }}"
      <span class="product_cost">{{ item.cost }}&nbsp;руб</span>
      <button class="btn btn-round">
        <a href="{% url 'basket:remove' item.pk %}" class="">
          удалить
        </a>
      </button>
    </div>
  {% endfor %}

  {% if basket_items %}
    <div class="basket_summary">
      В корзине {{ basket_items.0.total_quantity }} товаров общей стоимостью
      {{ basket_items.0.total_cost }} руб
    </div>
  {% endif %}

  <button class="btn btn-round">
    <a href="{% url 'main' %}">на главную</a>
  </button>
</div>
{% endblock %}
```

Для разнообразия мы не стали использовать механизм форм Django. Чтобы редактировать количество товаров в корзине в дальнейшем, создали элемент `<input type="number"...>` и ограничили его минимальное значение при помощи атрибута `min="0"`. Гиперссылка для удаления элемента уже не должна вызывать вопросов:

```
{% url 'basket:remove' item.pk %}
```

Если в вашей верстке не было страницы для просмотра корзины и стилей — потребуется некоторое время, чтобы их создать.

Теперь можно тестировать работу корзины.

Замечание:

Удаление в корзине реализовано через обычный URL-адрес. В перспективе необходимо добавить механизм подтверждения с формой и передачей данных методом **POST** (а значит, и защитой **CSRF**). Иначе третье лицо при помощи обычного GET-запроса сможет удалить все товары из вашей корзины.

Страница продукта

Есть рабочий модуль корзины. По клику в каталоге или на странице с «Горячим предложением» товар добавляется в нее. Логичнее было бы сделать, чтобы по клику на продукте открывалось его подробное описание. В рамках механизма CRUD это действие **Read**. Закрепим пройденный материал: создадим точку вызова контроллера **product()** в диспетчере URL приложения **mainapp**:

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('product/<int:pk>/', mainapp.product, name='product'),
]
```

Сам контроллер:

mainapp/views.py

```
def product(request, pk):
    title = 'продукты'

    content = {
        'title': title,
        'links_menu': ProductCategory.objects.all(),
        'product': get_object_or_404(Product, pk=pk),
        'basket': get_basket(request.user),
    }

    return render(request, 'mainapp/product.html', content)
```

Просто передаем объект продукта, полученный по **pk**, в шаблон:

mainapp/templates/mainapp/product.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="details-products">
    <div class="details-slider">
      <div class="slider-product">
        
      </div>
      <div class="slider-control">
        <div class="block">
          <a href="#">
            
          </a>
        </div>
        <div class="block">
          <a href="#">
            
          </a>
        </div>
        <div class="block">
          <a href="#">
            
          </a>
        </div>
      </div>
    </div>

    <div class="description">
      <h3 class="big-bold">{{ product.name }}</h3>
      <br>
      <p class="price"> {{ product.price }} <span>руб</span></p>
      <a href="{% url 'basket:add' product.pk %}" class="red-button">
        заказать
        <i class="fa fa-chevron-right" aria-hidden="true"></i>
      </a>
      <div class="description-text">
        {{ product.description }}
      </div>
    </div>
  </div>

</div>

<div class="clr"></div>

```

```
{% endblock %}
```

Этот шаблон является, по сути, копией шаблона **products.html**, только убрали раздел «Похожие продукты». Можно в перспективе объединить эти два шаблона в один или выделить общее и вынести в подшаблон, чтобы уменьшить количество повторяющегося кода.

Теперь меняем динамические адреса в шаблонах главной страницы и каталога:

```
{% url 'basket:add' product.pk %} -> {% url 'products:product' product.pk %}
```

Теперь при клике на продуктах они не будут добавляться в корзину, а станет появляться окно с подробным описанием продукта. Задача решена.

Вот преимущество динамических URL — мы сделали всего две или три правки в шаблонах, а поведение множества элементов сразу изменилось.

Декораторы: доступ только зарегистрированным

Исправим баг: когда незарегистрированный пользователь пытается купить товар, появляется сообщение об ошибке. Это будет непросто: будем вносить правки в нескольких приложениях и в файле конфигурации проекта.

Ограничим доступ к корзине только для зарегистрированных пользователей при помощи Django-декоратора [login_required](#):

basketapp/views.py

```
...

from django.contrib.auth.decorators import login_required
from django.urls import reverse

@login_required
def basket(request):
    ...
    return render(request, 'basketapp/basket.html', content)

@login_required
def basket_add(request, pk):
    if 'login' in request.META.get('HTTP_REFERER'):
        return HttpResponseRedirect(reverse('products:product', args=[pk]))

    ...
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))

@login_required
def basket_remove(request, pk):
    ...
    return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Вспомним, что декоратор — это, по сути, функция-обертка. И она может принимать дополнительные аргументы. В нашем случае это адрес, вызывающий контроллер аутентификации. Его можно задать двумя способами: как аргумент декоратора или в файле настроек. Выбираем второй способ и дописываем очередную константу в конец файла **settings.py**:

```
LOGIN_URL = '/auth/login/'
```

Теперь при попытке добавить товар в корзину Django проверит свойство **.is_authenticated** у объекта пользователя. Если оно **True** — товар добавится, иначе произойдет переход по адресу из константы **LOGIN_URL**.

* На этом этапе можно и остановиться, но мы хотим, чтобы после логина происходил возврат на страницу с товаром, который хотели купить. Это сложная задача, можете вернуться к ее решению позже.

При переходе на страницу логина, когда срабатывает декоратор **@login_required**, в GET-переменной **next** передается исходный адрес. Если хотели добавить в корзину товар с **pk=3**, в адресной строке браузера увидим:

```
http://127.0.0.1:8000/auth/login/?next=/basket/add/3/
```

Получим значение переменной **next** в контроллере **login()** и передадим в шаблон:

authapp/views.py

```
...

def login(request):
    title = 'Вход'

    login_form = ShopUserLoginForm(data=request.POST or None)

    next = request.GET['next'] if 'next' in request.GET.keys() else ''

    if request.method == 'POST' and login_form.is_valid():
        username = request.POST['username']
        password = request.POST['password']

        user = auth.authenticate(username=username, password=password)
        if user and user.is_active:
            auth.login(request, user)
            if 'next' in request.POST.keys():
                return HttpResponseRedirect(request.POST['next'])
            else:
                return HttpResponseRedirect(reverse('main'))

    content = {
        'title': title,
        'login_form': login_form,
        'next': next
    }

    return render(request, 'authapp/login.html', content)
```

Чтобы после ввода логина и пароля снова получить в контроллере значение переменной **next**, выведем его в шаблоне аналогично **CSRF** в виде скрытого поля формы **<input>**:

authapp/templates/authapp/login.html

```
...

{% csrf_token %}
{% if next %}
    <input type="hidden" name="next" value="{{ next }}">
{% endif %}
{{ login_form.as_p }}
...
```

Теперь, если мы логинимся при попытке добавить товар в корзину, сработает условие:

```
if 'next' in request.POST.keys():
```

Потому что в словаре **request.POST** будут все переменные формы, в том числе и **'next'**. Остается только перейти по сохраненному в **'next'** адресу: **HttpResponseRedirect(request.POST['next'])**.

Здесь нас ожидает сюрприз: после добавления товара в корзину контроллер **basket_add()** вернет нас на страницу логина:

```
return HttpResponseRedirect(request.META.get('HTTP_REFERER'))
```

Исправим этот баг:

basketapp/views.py

```
...

@login_required
def basket_add(request, pk):
    if 'login' in request.META.get('HTTP_REFERER'):
        return HttpResponseRedirect(reverse('products:product', args=[pk]))
    ...
```

Если мы попали в контроллер со страницы логина, сработает условие:

```
if 'login' in request.META.get('HTTP_REFERER'):
```

И вместо добавления товара перейдем на страницу продукта:

```
return HttpResponseRedirect(reverse('products:product', args=[pk]))
```

Здесь следует обратить внимание на работу с функцией **reverse()** в случае URL-адресов, передающих значения в контроллеры. Как и в шаблонах, надо это значение передать. Поэтому появился второй именованный аргумент **args=[pk]**.

Это обычный список значений, которые потребуются при создании URL-адреса.

Теперь все должно работать.

* AJAX: редактируем количество товаров в корзине

Можно по-разному организовать редактирование нескольких объектов модели на странице:

- при помощи Django FormSet — набора форм Django;
- при помощи JavaScript, синхронно;
- при помощи AJAX, асинхронно.

В настоящее время наблюдается устойчивая тенденция к применению технологии AJAX при разработке сайтов. Поэтому выберем третий способ.

Основное преимущество AJAX: по сети передаем только ту часть страницы, которая реально изменилась. Чтобы упростить реализацию AJAX, используем библиотеку [jQuery](#). Скачиваем, распаковываем и копируем в папку '**static/js/**'. Также создаем в этой папке файл '**basket_scripts.js**', в котором будут наши скрипты. Подключаем файлы в базовом шаблоне приложения **basketapp**:

basketapp/templates/basketapp/base.html

```
...
{% block js %}
    <script src="{% static 'js/jquery-3.2.1.min.js' %}"></script>
    <script src="{% static 'js/basket_scripts.js' %}"></script>
{% endblock %}
...
```

Содержимое файла **basket_scripts.js**:

static/js/basket_scripts.js

```
window.onload = function () {
    $('basket_list').on('click', 'input[type="number"]', function () {
        var t_href = event.target;

        $.ajax({
            url: "/basket/edit/" + t_href.name + "/" + t_href.value + "/",

            success: function (data) {
                $('basket_list').html(data.result);
            },
        });

        event.preventDefault();
    });
}
```

Здесь создали обработчик события '**click**' на элементах **<input>** типа **number** внутри DOM-элемента класса '**basket_list**'. Получаем источник события в переменную **t_href**:

```
var t_href = event.target
```

Важно понять, что **\$.ajax()** — это просто объект **jQuery**. Чтобы он сработал, необходимо передать ему объект, содержащий атрибуты и методы:

- **url** — атрибут, содержащий значение адреса, по которому необходимо выполнить запрос;
- **success** — метод, выполняющийся в случае успешного получения ответа от сервера.

В шаблоне будем атрибуту **'name'** элемента **<input>** присваивать значение **pk** продукта, а атрибуту **'value'** — текущее количество товаров в корзине. Тогда формируемый в скрипте URL-адрес будет иметь структуру:

```
/basket/edit/<product.pk>/<product.quantity>/'
```

Теперь необходимо добавить соответствующую строку в диспетчер URL приложения **basketapp**:

basketapp/urls.py

```
...  
  
path('edit/<int:pk>/<int:quantity>', basketapp.basket_edit, name='edit')  
  
...
```

Следующий шаг — создаем контроллер **basket_edit()**.

basketapp/views.py

```
...  
from django.template.loader import render_to_string  
from django.http import JsonResponse  
  
@login_required  
def basket_edit(request, pk, quantity):  
    if request.is_ajax():  
        quantity = int(quantity)  
        new_basket_item = Basket.objects.get(pk=int(pk))  
  
        if quantity > 0:  
            new_basket_item.quantity = quantity  
            new_basket_item.save()  
        else:  
            new_basket_item.delete()  
  
        basket_items = Basket.objects.filter(user=request.user).\  
            order_by('product__category')  
  
        content = {  
            'basket_items': basket_items,  
        }  
  
        result = render_to_string('basketapp/includes/inc_basket_list.html', \  
                                content)  
  
        return JsonResponse({'result': result})
```

Чтобы контроллер работал только для запросов, переданных методом AJAX — делаем проверку:


```
if request.is_ajax():
```

Если количество товара стало равно нулю — удаляем его из корзины.

После изменений, делаем запрос и получаем обновленное содержимое корзины в переменной **basket_items**.

Важно: шаблон рендерим в строку при помощи функции **render_to_string()** из модуля **django.template.loader**.

Ответ отправляем в привычном для AJAX формате JSON при помощи функции **JsonResponse()** из модуля **django.http**.

AJAX получает ответ и при помощи метода **html()** заменяет содержимое DOM-элемента с классом **'basket_list'**.

При работе с AJAX приходится продумывать структуру шаблонов. В нашем случае содержимое корзины вынесли в отдельный подшаблон:

basketapp/templates/basketapp/includes/inc_basket_list.html

```
{% for item in basket_items %}
    <div class="basket_record">
        
        <span class="category_name">{{ item.product.category.name }}</span>
        <span class="product_name">{{ item.product.name }}</span>
        <span class="product_price">{{ item.product.price }}&nbsp;руб</span>
        <input type="number" name="{{ item.pk }}" min="0"
            value="{{ item.quantity }}">
        <span class="product_cost">{{ item.cost }}&nbsp;руб</span>
        <button class="btn btn-round">
            <a href="{% url 'basket:remove' item.pk %}" class="">
                удалить
            </a>
        </button>
    </div>
{% endfor %}

{% if basket_items %}
    <div class="basket_summary">
        В корзине {{ basket_items.0.total_quantity }} товаров общей стоимостью
        {{ basket_items.0.total_cost }} руб
    </div>
{% endif %}
```

При этом шаблон страницы корзины стал совсем простым:

basketapp/templates/basketapp/basket.html

```
{% extends 'basketapp/base.html' %}
{% load staticfiles %}

{% block content %}
    <div class="basket_list">
        {% include 'basketapp/includes/inc_basket_list.html' %}
    </div>
    <button class="btn btn-round">
        <a href="{% url 'main' %}">
            на главную
        </a>
    </button>
{% endblock %}
```

Каждый раз при выполнении запроса AJAX будет обновляться только содержимое внутри блока:

```
<div class="basket_list">
    ...
</div>
```

Если бы мы разместили кнопку со ссылкой «На главную» внутри этого блока — при первом же срабатывании AJAX она бы исчезла. Попробуйте и подумайте, почему так происходит.

Практическое задание

1. Добавить к модели корзины методы и вывести в меню количество товаров и их полную стоимость.
2. Реализовать механизм просмотра содержимого корзины и удаления товаров из нее.
3. Реализовать просмотр товара, скорректировать адреса в каталоге и на главной странице так, чтобы при нажатии на товар появлялась страница просмотра. Добавление товара в корзину теперь должно быть только с этой страницы.
4. Защитить доступ к корзине декоратором **@login_required**.
5. * Реализовать асинхронное редактирование количества товаров в корзине при помощи AJAX.
6. * Реализовать механизм вывода случайного товара на странице «Горячее предложение», которая появляется при входе в каталог.

Дополнительные материалы

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

1. [Объект QuerySet.](#)
2. [Запросы при помощи ORM.](#)
3. [Property in models.](#)
4. [Система аутентификации Django \(декораторы\).](#)
5. [Auth decorators.](#)
6. [jQuery.](#)
7. [jQuery+AJAX.](#)