



Урок 5

Пользователь + товар = корзина

Выполнение запросов при помощи ORM. Работа с меню.
Создание корзины.

[Работа с запросами в Django ORM](#)

[Выполнение запросов в консоли](#)

[Вывод товаров выбранной категории](#)

[Работа с меню категорий](#)

[Корзина](#)

[Модель](#)

[Диспетчер URL](#)

[Контроллеры](#)

[Корректировка шаблонов с товарами](#)

[Добавление счетчика купленных продуктов в меню](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Работа с запросами в Django ORM

На 3 уроке мы научились передавать в контроллер номер выбранной категории (pk). Но на контенте страницы это никак не отражалось. Сегодня мы это исправим. Сначала немного теории.

Выполнение запросов в консоли

Запустим консоль в корне проекта (`python manage.py shell`) и выполним код:

```
>>>from mainapp.models import ProductCategory, Product
>>>products = Product.objects.filter(category__pk=1)
```

Если сейчас посмотреть содержимое переменной `products` — увидим список товаров в категории, значение первичного ключа которой в базе равно 1.

Замечание: напоминаем, что `pk` (первичный ключ) и `id` (идентификатор товара) — это одно и то же в нашем проекте.

В ORM выполнение запросов происходит через методы `filter()`, `exclude()` и `get()` менеджера модели. Аргументы методов именованные — это имена атрибутов модели. При обращении к атрибуту связанной модели — пишем его через двойное подчеркивание после имени атрибута текущей модели, через который она подключена. Например:

- `category__pk` — обращаемся к атрибуту `pk` связанной модели `ProductCategory`;
- `category__name` — обращаемся к атрибуту `name` связанной модели `ProductCategory`;
- `category__description` — обращаемся к атрибуту `description` связанной модели `ProductCategory`.

Не забывайте, что в результате работы менеджера модели в большинстве случаев мы получаем объект [QuerySet](#), который по сути является списком, но имеет некоторые дополнительные атрибуты и методы (например, атрибут `.query`, хранящий текст запроса).

Выполним еще один запрос:

```
>>>product1 = Product.objects.filter(pk=1)
```

Теперь получим объект `QuerySet`, состоящий из одного элемента — продукта, у которого `pk=1`. Эквивалентный SQL-запрос имел бы вид:

```
SELECT * FROM mainapp_product WHERE pk=1
```

Можем посмотреть реальный запрос, который сформировал Django:

```
>>>print(product1.query)
```

На самом деле не стоит этим увлекаться: основная идея Django ORM — абстрагироваться от языка запросов SQL.

Вы уже обратили внимание, что метод `filter()` всегда возвращает список — даже если искали уникальную запись в базе. В таких случаях удобнее воспользоваться методом `get()`:

```
>>>product_1 = Product.objects.get(pk=1)
```

Тут мы сразу получим в переменной объект продукта.

Замечание:

1. В данном случае текст запроса получить не удастся — ведь нам вернулся не `QuerySet`, а объект модели `Product`.
2. Если объект не найден — будет сгенерирована ошибка [DoesNotExist](#), которую необходимо обработать в коде.

Чуть не забыли про метод `exclude()` — он эквивалентен логическому отрицанию при выполнении запроса: «найти все записи, кроме тех, что удовлетворяют условию в скобках». Например:

```
>>>products = Product.objects.exclude(category__name='дом')
```

Найдет продукты всех категорий, кроме категории «дом».

Хорошая новость — к объекту `QuerySet` можно применять методы менеджера модели. Это значит, что мы можем выстраивать цепочки запросов:

```
Product.objects.exclude(category__name='дом').filter(price__gte=4500.99)
```

Получим продукты всех категорий, кроме категории «дом», у которых цена больше или равна 4500,99 руб.

Как вы уже догадались, [модификаторы](#) сравнения прописываются тоже через двойное подчеркивание после имени атрибута модели:

- `in` — принадлежность списку (`in`);
- `gt` — больше (`greater`);
- `gte` — больше или равно (`greater than or equal to`);
- `lt` — меньше (`less`);
- `lte` — меньше или равно (`less than or equal to`);
- `startswith` — регистрозависимое «начинается с» (`case-sensitive`);
- `endswith` — регистрозависимое «заканчивается на» (`case-sensitive ends-with`).

Логический оператор «И» в SQL-запросе можно реализовать, просто прописав аргументы через запятую:

```
Product.objects.filter(price__lte=4500.99, quantity__gte=3)
```

Получим продукты, цена которых не превышает 4500,99 руб. и остаток на складе не менее 3 штук.

Осталось отсортировать результат запроса — это делается при помощи метода `order_by()`:

```
Product.objects.filter(quantity__gt=5).order_by('name')
```

Реверс сортировки задается знаком '-' перед именем атрибута:

```
Product.objects.filter(quantity__gt=5).order_by('-name', 'price')
```

Здесь мы сделали сортировку по двум атрибутам: для 'name' — по убыванию, а для 'price' — по возрастанию.

Ограничить количество объектов в запросе можно как обычно для списков — при помощи срезов. Отрицательные индексы, например, [-1] — не поддерживаются.

И, самое главное: **запросы** в Django ORM **ленивы** — они выполняются только при **реальном** обращении к данным (вывод на экран или обработка).

Все. Теперь смело можно решать боевую задачу.

Вывод товаров выбранной категории

Доработаем контроллер `products()` в приложении `mainapp`:

`mainapp/views.py`

```
...
from django.shortcuts import get_object_or_404
...
def products(request, pk=None):
    print(pk)

    title = 'продукты'
    links_menu = ProductCategory.objects.all()

    if pk:
        if pk == '0':
            category = {'name': 'все'}
            products = Product.objects.all().order_by('price')
        else:
            category = get_object_or_404(ProductCategory, pk=pk)
            products = Product.objects.filter(category__pk=pk).order_by('price')

    content = {
        'title': title,
        'links_menu': links_menu,
        'category': category,
        'products': products,
    }

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

same_products = Product.objects.all()[3:5]

content = {
    'title': title,
```

```

        'links_menu': links_menu,
        'same_products': same_products
    }

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

```

Если аргумент `pk` не был передан диспетчером URL (мы только зашли на страницу с каталогом и еще не выбрали категорию в подменю), наш контроллер ведет себя, как раньше.

Если же пришло любое число — сработает код внутри условия «`if pk:`». Даже если это значение '0'! Потому, что аргумент `pk` имеет тип `str` — мы же взяли его из url-адреса, который является строкой.

Следует пояснить [функцию](#) `get_object_or_404()` из модуля `django.shortcuts`. Это обертка над методом `get()` менеджера модели. Мы уже говорили, что этот метод может сгенерировать ошибку. Есть два варианта: либо обработать ее в контроллере вручную, либо поручить это Django (как мы и сделали).

Остальная часть кода должна быть понятна: полученный по значению ключа `pk` объект категории и список ее продуктов передаем в шаблон `'mainapp/products_list.html'` и рендерим. Если передан '0' — создаем псевдо-категорию «все» и получаем список всех продуктов магазина.

Также мы всегда передаем в шаблон список объектов категорий `links_menu`, чтобы позже реализовать автоматическую генерацию меню.

Теперь как обычно создаем шаблон.

`mainapp/templates/mainapp/products_list.html`

```

{% extends 'mainapp/base.html' %}
{% load staticfiles %}

{% block menu %}
    <div class="hero-white">
        <div class="header clearfix">
            {% include 'mainapp/includes/inc_menu.html' %}
        </div>
    </div>
{% endblock %}

{% block content %}
    <div class="details">
        <div class="links clearfix">
            {% include 'mainapp/includes/inc_categories_menu.html' %}
        </div>

        <div class="products_list">
            <div class="title clearfix">
                <h2>
                    Категория: "{{ category.name|title }}"
                </h2>
            </div>
            <div class="category-products clearfix">

```

```

    {% for product in products %}
    <div class="block">
      <a href="#">
        
        <div class="text">
          
          <h4>{{ product.name }}</h4>
          <p>{{ product.description }}</p>
        </div>
      </a>
    </div>
    {% endfor %}

  </div>
</div>
</div>

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

{% endblock %}

```

В коде шаблона нет ничего нового. Просто выводим переданные из контроллера элементы контекста.

Работа с меню категорий

Прежде, чем двигаться дальше, давайте реализуем механизм автоматической генерации меню категорий. Мы уже передали в шаблон список объектов категорий `links_menu`. Давайте изменим код подшаблона:

mainapp/templates/mainapp/includes/inc_categories_menu.html

```

<ul class="links-menu">
  <li>
    <a href="{% url 'products:category' 0 %}"
      class="{% if request.resolver_match.kwargs.pk == '0' %}
        active
      {% endif %}">
      Все
    </a>
  </li>
  {% for link in links_menu %}
  <li>
    <a href="{% url 'products:category' link.pk %}"
      class="{% if request.resolver_match.kwargs.pk|add:'0' == link.pk %}
        active
      {% endif %}">
      {{ link.name }}
    </a>
  </li>
  {% endfor %}
</ul>

```

Тут много нового.

Во-первых, при работе с адресами, из которых диспетчер URL должен извлекать данные (в нашем случае, номер категории) — необходимо эти данные в шаблоне передать:

```
{% url 'products:category' link.pk %}
```

Как читать этот тег? Адрес с именем `category` из пространства имен `products`, соответствующий категории с номером `link.pk`. Для псевдо-категории «все» прописываем вручную номер '0'.

*Во-вторых, сразу реализуем работу с классом `active`. Получаем именованный аргумент, переданный в url-адресе:

```
request.resolver_match.kwargs.pk
```

Как и в Python, `kwargs` — означает KeyWord ARGumentS. Имя аргумента: `pk`. Если вы просто попытаетесь сравнить его значение с `pk` категории (`link.pk`) — ничего не получится. Все дело в том, что `request.resolver_match.kwargs.pk` возвращает строковое значение, а переменная `link.pk` — это число. Мы применили небольшой лайфхак: фильтр `add: '0'`. По сути это просто «плюс ноль», но при этом меняется тип со строки на число (помните в Python неявное преобразование строки в число при сложении?). Но это не все: необходимо внести изменения в диспетчере URL приложения `mainapp`:

`mainapp/urls.py`

```
from django.conf.urls import url

import mainapp.views as mainapp

urlpatterns = [
    url(r'^$', mainapp.products, name='index'),
    url(r'^category/(?P<pk>\d+)/$', mainapp.products, name='category'),
]
```

Имя `pk` аргументу в гиперссылке задается благодаря, тому, что дописали `?P<pk>` в регулярном выражении. В будущем нам еще не раз понадобится эта возможность.

Теперь можно проверять: должны выводиться товары в категориях. Но мы обнаружим баг: при выборе элемента «продукты» основного меню — он не подсвечивается (возможно, вы еще раньше это заметили). Исправим это — просто скорректируем подшаблон основного меню.

`mainapp/templates/mainapp/includes/inc_menu.html`

```
...
<li>
  <a href="{% url 'products:index' %}"
    class="{% if request.resolver_match.namespace == 'products' %}
      active
    {% endif %}">
    продукты
  </a>
</li>
```

```
...
```

Здесь мы заменили у `request.resolver_match` атрибут `url_name` на `namespace`. Причина — мы же перешли на пространство имен `namespace` для этого адреса. Теперь класс `active` снова работает, когда выбираем пункт основного меню «продукты».

В очередной раз напомним, что это материал повышенной сложности — можете вернуться к нему позже.

Корзина

Корзину можно реализовать в магазине несколькими способами:

- через сессии — хранить товары, выбранные пользователем в сессии;
- через JS — создать объект корзины в скрипте и работать с ним на стороне клиента;
- через БД — хранить товары корзины в базе данных.

Возможны и другие варианты. Все они имеют преимущества и недостатки. Мы в проекте реализуем вариант с БД — создадим модель корзины и обеспечим работу с ней при помощи контроллеров. Логично этот код вынести в отдельное приложение, чтобы можно было использовать в других проектах.

Итак, создаем новое приложение `basketapp` и сразу прописываем доступ к его контроллерам через пространство имен `basket` в главном диспетчере URL:

`geekshop/urls.py`

```
...
url(r'^basket/', include('basketapp.urls', namespace='basket')),
...
```

Модель

В приложении `basketapp` создаем модель корзины:

`basketapp/models.py`

```
from django.db import models
from django.conf import settings
from mainapp.models import Product

class Basket(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(verbose_name='количество', default=0)
    add_datetime = models.DateTimeField(verbose_name='время', auto_now_add=True)
```

Очень важный момент: раз мы используем **свою** модель пользователя — именно ее необходимо связать с моделью корзины. Для этого сначала импортируем файл настроек:

```
from django.conf import settings
```


Прописываем первым аргументом в `models.ForeignKey`:

```
models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
```

Теперь, если вы измените модель пользователя магазина — автоматически изменится и связь в корзине.

Вторая связанная модель в корзине — модель продукта.

Обратите внимание на значение аргумента `auto_now_add=True` атрибута `add_datetime` корзины — это позволяет автоматически фиксировать дату и время добавления товара.

Если прямо сейчас выполнить миграции — будет ошибка. Почему? Правильно: необходимо прописать имя нового приложения в файл настроек проекта. А теперь можно пробовать. Все должно пройти без ошибок. Убедитесь, что в БД появилась таблица `basketapp_basket`.

В дальнейшем необходимо будет добавить в модель корзины методы, позволяющие подсчитать сумму покупок и их количество.

Диспетчер URL

Прежде чем писать контроллеры, создадим их точки вызова в диспетчере URL:

`basketapp/urls.py`

```
from django.conf.urls import url

import basketapp.views as basketapp

urlpatterns = [
    url(r'^$', basketapp.basket, name='view'),
    url(r'^add/(?P<pk>\d+)/$', basketapp.basket_add, name='add'),
    url(r'^remove/(?P<pk>\d+)/$', basketapp.basket_remove, name='remove'),
]
```

Здесь мы, как обычно при работе с моделями, реализуем механизм CRUD:

- `basket()` — просмотр и редактирование корзины (Read & Update);
- `basket_add()` — добавление товара в корзину (Create);
- `basket_remove()` — удаление товара из корзины (Delete).

Будем привыкать передавать контроллерам именованные аргументы `(?P<pk>\d+)`.

Контроллеры

Для начала решим задачу добавления товара в корзину, а остальные контроллеры сделаем заглушками:

basketapp/views.py

```
from django.shortcuts import render, HttpResponseRedirect, get_object_or_404
from basketapp.models import Basket
from mainapp.models import Product

def basket(request):
    content = {}
    return render(request, 'basketapp/basket.html', content)

def basket_add(request, pk):
    product = get_object_or_404(Product, pk=pk)
    old_basket_item = Basket.objects.filter(user=request.user, product=product)

    if old_basket_item:
        old_basket_item[0].quantity += 1
        old_basket_item[0].save()
    else:
        new_basket_item = Basket(user=request.user, product=product)
        new_basket_item.quantity += 1
        new_basket_item.save()

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

def basket_remove(request):
    content = {}
    return render(request, 'basketapp/basket.html', content)
```

Получаем объект продукта из базы данных. Проверяем, есть ли он в корзине текущего пользователя:

```
Basket.objects.filter(user=request.user, product=product)
```

Увеличиваем счетчик либо у существующего объекта корзины, либо у созданного (в модели всегда задается значение счетчика '0' по умолчанию!). Обязательно сохраняем объект.

Чтобы вернуться на ту же страницу, где добавляли товар, мы используем очередную хитрость:

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

Можно прочитать это так: «вернуться туда же, откуда пришли». На самом деле при более глубоком погружении в Django, словарь `request.META` может быть очень полезен — он содержит параметры http-запроса. Мы использовали стандартный метод `get()` языка Python, чтобы получить значение ключа `'HTTP_REFERER'`. Можно было это сделать и при помощи квадратных скобок.

Корректировка шаблонов с товарами

Чтобы все заработало, необходимо на страницах с товарами магазина прописать адреса, вызывающие контроллер `basket_add()`. Например:

`mainapp/products_list.html`

```
...
<div class="category-products clearfix">
  {% for product in products %}
    <div class="block">
      <a href="{% url 'basket:add' product.pk %}">
        
        <div class="text">
          
          <h4>{{ product.name }}</h4>
          <p>{{ product.description }} </p>
        </div>
      </a>
    </div>
  {% endfor %}
</div>
...
```

Теперь можно пробовать. При клике на товары каталога, в базе должны появляться записи. Внешних изменений на странице происходить не должно.

Один **нюанс**: вы должны быть **залогинены** на сайте, иначе — ошибка. В дальнейшем мы при помощи **декораторов** решим эту проблему. Пока просто будьте аккуратны — входите в систему перед работой с корзиной.

Добавление счетчика купленных продуктов в меню

Пришла пора оживить иконку корзины в меню. Сделаем простейший счетчик количества купленных продуктов для страницы «продукты» при помощи шаблонного фильтра. Для этого необходимо создать в контроллере и передать в шаблон объект корзины:

mainapp/views.py

```
...
    basket = []
    if request.user.is_authenticated:
        basket = Basket.objects.filter(user=request.user)

    if pk:
        if pk == '0':
            products = Product.objects.all().order_by('price')
            category = {'name': 'все'}
        else:
            category = get_object_or_404(ProductCategory, pk=pk)
            products = Product.objects.filter(category__pk=pk).order_by('price')

    content = {
        'title': title,
        'links_menu': links_menu,
        'category': category,
        'products': products,
        'basket': basket,
    }

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

По умолчанию создаем пустую корзину (`basket = []`). Если пользователь в системе (`request.user.is_authenticated`), получаем все его записи из модели корзины:

```
basket = Basket.objects.filter(user=request.user)
```

И передаем в контекст.

Остается доработать подшаблон основного меню:

mainapp/templates/mainapp/includes/inc_menu.html

```
<a href="{% url 'main' %}" class="logo"></a>
<ul class="menu">
...
</ul>
<a href="#" class="search"></a>
<a href="{% url 'basket:view' %}" class="basket">
    <span>
        {% if basket %} {{ basket|length }} {% endif %}
    </span>
</a>
```

Шаблонный фильтр `length` эквивалентен функции `len()` языка Python. Поэтому мы получаем не общее количество продуктов в корзине, а количество записей в БД для текущего пользователя — это число видов продуктов.

Не забывайте прописать стили для новых элементов страниц!

Можете проверить — при многократном нажатии на продукте, счетчик (поле `quantity` в БД) увеличивается, а число продуктов корзины в меню — нет. Оно увеличивается на единицу только при первом добавлении товара.

Домашнее задание

1. Поработать с запросами в **консоли** через механизм Django ORM.
2. Реализовать механизм вывода товаров по категориям.
3. *Организовать динамическую генерацию меню категорий и подсветку выбранной категории.
4. Создать приложение корзины.
5. Реализовать механизм добавления товара в корзину.
6. Вывести в меню счетчик купленных товаров.
7. *Написать в модели корзины методы для определения общего количества и стоимости добавленных товаров и вывести эти величины в меню вместо счетчика, сделанного на уроке.

Дополнительные материалы

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

1. [Объект QuerySet](#)
2. [Запросы при помощи ORM](#)
3. [request.META](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Django Book\(rus\)](#)