Урок 7

Еще быстрее: кеширование в Django

Декоратор «@cached_property»: кешируем методы моделей.

Тег «With» – кешируем переменные в шаблонах.

Кешируем функции, фрагменты шаблона и контроллеры при помощи «Memcached»

Введение

Встроенные механизмы кеширования

- Кеширование вычисляемых полей моделей
- Кеширование в шаблонах при помощи тега «with»

Кеширование при помощи Memcached

- Установка и настройка Memcached
- Низкоуровневое кеширование
- Кеширование фрагментов шаблона
- Кеширование контроллеров
- Кеширование всего сайта

Практическое задание

Дополнительные материалы

Используемая литература

Введение

На прошлом уроке мы научились оценивать производительность проекта. За счет правильной работы с запросами уменьшили избыточность и повысили скорость работы. Чтобы все работало еще быстрее - добавим кеширование. При этом очень важно понимать, что всегда есть противоречие между скоростью и актуальностью возвращаемого контента. Это особенно важно для интернет-магазина, когда содержимое страницы меняется очень быстро (число товаров в заказе или корзине).

В Django есть встроенные механизмы, которые позволяют кешировать методы моделей и переменные при рендеринге шаблонов. Если нам нужно хранить объекты или результаты работы контроллеров - можно организовать кеш в оперативной памяти, на жестком диске или в базе данных.

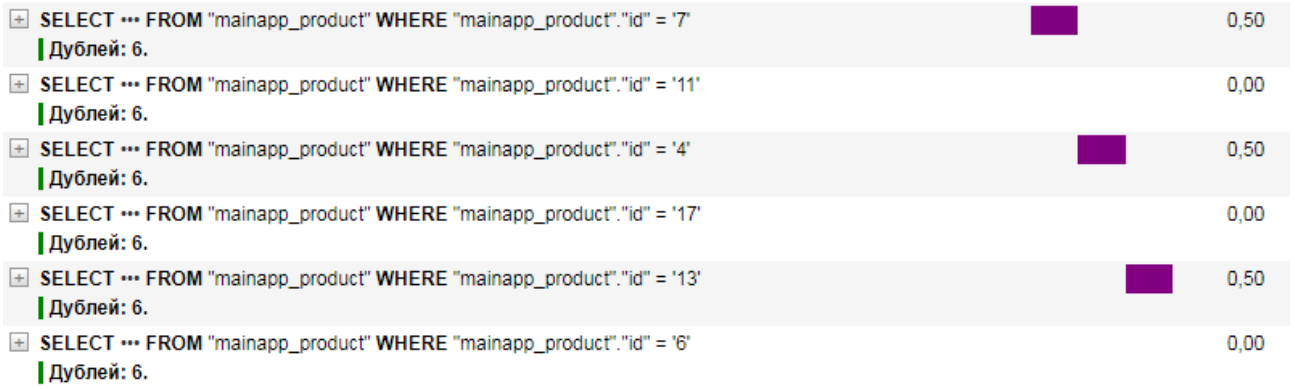
Встроенные механизмы кеширования

Кеширование вычисляемых полей моделей

Как мы уже говорили ранее, в Django рекомендуется делать «жирные» модели - это значит по возможности переносить методы из контроллеров в модели. Кроме того, что уменьшается количество повторяющегося кода, у этого подхода есть еще одно преимущество: можно кешировать значения вычисляемых значений при помощи декоратора «cached_property» из модуля «django.utils.functional». Кешировать можно *только* методы с единственным аргументом «self». При этом метод становится атрибутом модели - необходимо переписывать код, где он вызывался. Значение хранится до его обновления или до тех пор пока существует объект модели.

Область применения декоратора - ситуации, когда метод вызывается несколько раз за время существования объекта - например, сначала в контроллере, а затем в шаблоне, или при выполнении цепочки методов.

Попробуем уменьшить число дублей запросов при редактировании корзины. При помощи вкладки «SQL» отладочной панели проанализируем вывод корзины пользователя «user1». Всего выполняется 12 запросов, из них 8 - дубликаты. Шесть дубликатов появились в результате выполнения метода «_get_product_cost()» при вычислении стоимости продуктов данного вида:



Если кешировать результаты запроса продуктов корзины пользователя - эти дубликаты должны исчезнуть. Перепишем код модели:

geekshop/basketapp/models.py

```
...
from django.utils.functional import cached_property

class Basket(models.Model):
    ...
    @cached_property
    def get_items_cached(self):
        return self.user.basket.select_related()

    def get_total_quantity(self):
        _items = self.get_items_cached
        return sum(list(map(lambda x: x.quantity, _items)))

    def get_total_cost(self):
        _items = self.get_items_cached
        return sum(list(map(lambda x: x.product_cost, _items)))
```

Мы создали метод «get_items_cached()» и кешировали его помощи декоратора «@cached_property». В результате получили одноименный атрибут, который используем в методах «get_total_quantity()» и «get_total_cost()». При проверке можно увидеть, что число дубликатов уменьшилось на 6 - кеш работает.

Проведем для адреса «http://192.168.0.98/basket/» тест с кешированием и без:

```
siege -f /home/django/geekshop/urls.txt -d0 -r20 -c125
```

Для корзины, содержащей 16 товаров пяти видов время выполнения уменьшается примерно на 6%.

Кеширование в шаблонах при помощи тега «with»

Рассмотрим еще один способ кеширования в Django - использование в шаблонах тега «{% with … %}». Если у нас есть ресурсоемкое значение (например, вычисляемое поле), которое используется несколько раз в шаблоне, то можно его сохранить в некоторой переменной и уменьшить количество вызовов до одного.

Рассмотрим на примере задачи, которую мы только что решали: вычисление количества продуктов и стоимости, но уже для заказа. Вместо двух методов «get_total_quantity()» и «get_total_cost()» создадим один «get_summary()», возвращающий словарь:

geekshop/ordersapp/models.py

```
...
class Order(models.Model):
    ...

    def get_summary(self):
        items = self.orderitems.select_related()
        return {
```

```
            'total_cost': sum(list(map(lambda x: x.quantity * x.product.price,\
                                                                    items))),
            'total_quantity': sum(list(map(lambda x: x.quantity, items)))
        }
```

В шаблоне теперь можно записать:

geekshop/ordersapp/templates/ordersapp/includes/inc_order_summary.html

```
{% if object %}
   ...
   <hr>
   {% with object_summary=object.get_summary %}
       <div class="h4">
           общее количество товаров:
           <span class="order_total_quantity">
               {{ object_summary.total_quantity }}
           </span>
       </div>
       <div class="h3">
           общая стоимость:
           <span class="order_total_cost">
               {{ object_summary.total_cost }}
           </span> руб
       </div>
   {% endwith %}
...
```

В данном случае большого эффекта от такой оптимизации ждать не стоит - вычислительная сложность кешированных нами полей невысокая. Если провести тест для контроллеров редактирования заказа - получим прирост производительности около 4%. В других ситуациях он может быть существеннее.

Кеширование при помощи Memcached

Для дальнейшего повышения скорости работы проекта организуем кеширование объектов при помощи популярного и проверенного решения - Memcached. По сути, мы получаем словарь, в котором можно хранить большие объемы данных под соответствующими ключами.

Установка и настройка Memcached

Устанавливаем в виртуальное окружение:

```
sudo apt install memcached
sudo apt install libmemcached-dev
pip install python-memcached
```

Открываем файл настроек:

```
sudo nano /etc/memcached.conf
```

Важные настройки:

- **«-m 256»** - объем памяти, выделяемой под кеш, Мб;
- **«-p 11211»** - порт для соединения.

Чтобы применить все изменения, необходимо перезапустить службу:

```
sudo systemctl restart memcached
```

Проверить ситуацию можно, выполнив команду:

```
ps aux | grep memcached
```

Дополнительный функционал Django реализуется при помощи соответствующего бекэнда. Подключаем в настройках проекта:

geekshop/geekshop/settings.py

```
...
if os.name == 'posix':
    CACHE_MIDDLEWARE_ALIAS = 'default'
    CACHE_MIDDLEWARE_SECONDS = 120
    CACHE_MIDDLEWARE_KEY_PREFIX = 'geekshop'

    CACHES = {
        'default': {
            'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
            'LOCATION': '127.0.0.1:11211',
        }
    }

LOW_CACHE = True
```

В Django можно использовать несколько кешей - их бэкенды должны быть в словаре CACHES. У нас один кеш - записали его в ключ «default».

Важно: dev-сервер Django по умолчанию использует кеширование в оперативной памяти при помощи бекэнда «django.core.cache.backends.locmem.LocMemCache».

В константе «CACHE_MIDDLEWARE_ALIAS» указываем, какой из кешей использовать по умолчанию. Длительность хранения данных в кеше - константа «CACHE_MIDDLEWARE_SECONDS». Ее значение необходимо настраивать в зависимости от особенностей вашего проекта. Так как в перспективе на одном сервере может быть несколько ресурсов - сразу зададим префикс для ключей кеша нашего проекта: «CACHE_MIDDLEWARE_KEY_PREFIX».

Для самого бекэнда «MemcachedCache» настроим адрес и порт: 127.0.0.1:11211.

Константу «LOW_CACHE» используем для облегчения тестирования: она позволит включать или выключать низкоуровневое кеширование в проекте.

В Django кеширование настраивается достаточно гибко:

- кеширование всего сайта;
- кеширование контроллеров;
- кеширование фрагментов шаблона;
- низкоуровневое кеширование.

Низкоуровневое кеширование

Начнем с самого низкого уровня. С одной стороны, он требует больше всего усилий для реализации, но с другой - здесь можно решить задачи, недоступные на более абстрактных уровнях.

Добавим в код контроллеров приложения «mainapp» функции:

geekshop/mainapp/views.py

```
...
from django.conf import settings
from django.core.cache import cache

def get_links_menu():
    if settings.LOW_CACHE:
        key = 'links_menu'
        links_menu = cache.get(key)
        if links_menu is None:
            links_menu = ProductCategory.objects.filter(is_active=True)
            cache.set(key, links_menu)
        return links_menu
    else:
        return ProductCategory.objects.filter(is_active=True)


def get_category(pk):
    if settings.LOW_CACHE:
        key = f'category_{pk}'
        category = cache.get(key)
        if category is None:
            category = get_object_or_404(ProductCategory, pk=pk)
            cache.set(key, category)
        return category
    else:
        return get_object_or_404(ProductCategory, pk=pk)


def get_products():
```

```
    if settings.LOW_CACHE:
        key = 'products'
        products = cache.get(key)
        if products is None:
            products = Product.objects.filter(is_active=True, \
                        category__is_active=True).select_related('category')
            cache.set(key, products)
        return products
    else:
        return Product.objects.filter(is_active=True, \
                        category__is_active=True).select_related('category')


def get_product(pk):
    if settings.LOW_CACHE:
        key = f'product_{pk}'
        product = cache.get(key)
        if product is None:
            product = get_object_or_404(Product, pk=pk)
            cache.set(key, product)
        return product
    else:
        return get_object_or_404(Product, pk=pk)


def get_products_orederd_by_price():
    if settings.LOW_CACHE:
        key = 'products_orederd_by_price'
        products = cache.get(key)
        if products is None:
            products = Product.objects.filter(is_active=True, \
                                category__is_active=True).order_by('price')
            cache.set(key, products)
        return products
    else:
        return Product.objects.filter(is_active=True,\
                                category__is_active=True).order_by('price')


def get_products_in_category_orederd_by_price(pk):
    if settings.LOW_CACHE:
        key = f'products_in_category_orederd_by_price_{pk}'
        products = cache.get(key)
        if products is None:
            products = Product.objects.filter(category__pk=pk, is_active=True,\
                                category__is_active=True).order_by('price')
            cache.set(key, products)
        return products
    else:
        return Product.objects.filter(category__pk=pk, is_active=True, \
                                category__is_active=True).order_by('price')
```

Из модуля «django.core.cache» импортируем объект «cache», который дает нам доступ к кешу по умолчанию (ключ «default» в словаре CACHES). Если вы будете использовать несколько бекэндов для кеширования - лучше импортировать словарь «caches» и брать каждый раз из него по ключу d соответствующий кеш.

Для доступа к созданной нами константе «LOW_CACHE» в файле настроек проекта, мы импортировали «settings из django.conf».

Главная задача при работе с кешем на низком уровне - спланировать имена ключей. Мы используем первичный ключ «pk» для уникальности продуктов и категорий. Если значение переменной уникально для каждого пользователя - будем добавлять его «pk» к ключу.

Максимальный размер ключа в «Memcached» - 250 символов (250 байт). Допустимо использовать только латиницу и цифры. Без пробелов и прочих спецсимволов.

Находим в контроллерах точки вызова соответствующих значений и меняем их на наши функции. Также можем добавить кеширование в контроллер «contact()»:

geekshop/mainapp/views.py

```
...
def get_hot_product():
    products = get_products()

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

def main(request):
    title = 'главная'
    products = get_products()[:3]
    ...

def products(request, pk=None, page=1):
    title = 'продукты'
    links_menu = get_links_menu()

    if pk:
        if pk == '0':
            ...
            products = get_products_orederd_by_price()
        else:
            category = get_category(pk)
            products = get_products_in_category_orederd_by_price(pk)

        ...
    hot_product = get_hot_product()
    same_products = get_same_products(hot_product)
    ...

def product(request, pk):
    title = 'продукты'
    links_menu = get_links_menu()
    product = get_product(pk)
    ...
```

```
def contact(request):
    title = 'о нас'
    if settings.LOW_CACHE:
        key = f'locations'
        locations = cache.get(key)
        if locations is None:
            locations = load_from_json('contact__locations')
            cache.set(key, locations)
    else:
        locations = load_from_json('contact__locations')
    ...
```

Для оценки эффективности реализованного кеширования оставим в списке адресов для теста «siege» только следующие:

urls.txt

```
http://192.168.0.98/
http://192.168.0.98/products/
http://192.168.0.98/contact/
http://192.168.0.98/products/category/1/
http://192.168.0.98/products/category/2/
http://192.168.0.98/products/category/3/
http://192.168.0.98/products/category/4/
```

Проведем тесты с кешированием и без:

```
siege -i -f /home/django/geekshop/urls.txt -d0 -r17 -c125
```

В результате обнаружим, что время выполнения уменьшилось на 5%: с 37,41 до 35.53 сек. Следует отметить, что мы кешировали только **очень редко** меняющиеся данные. В будущем эффективность можно повысить, добавив кеширование динамических величин с их обновлением по сигналам или другим событиям.

Кеширование фрагментов шаблона

Для дальнейшего повышения производительности попробуем кешировать в шаблоне поле формы со списком продуктов. По данным панели «TemplateProfiler» его рендеринг занимает достаточно много времени:

geekshop/ordersapp/templates/ordersapp/order_form.html

```
{% load cache %}
...
{% for field in form.visible_fields %}
    <td class="{% cycle 'td1' 'td2' 'td3' 'td4' %} order formset_td">
        ...
        {% if field.name != 'price' %}
            {% if field.name == 'product' %}
```

```
                {% cache 3600 orderitemform_product field.value %}
                    {{ field }}
                {% endcache %}
            {% else %}
                {{ field }}
            {% endif %}
        {% else %}
            ...
{% endfor %}
...
```

Важно: чтобы работать с кешем в шаблоне, его необходимо загрузить:

```
{% load cache %}
```

В цикле по полям формы сохраняем в кеше шаблона поле «product» в уникальный ключ с префиксом «orderitemform_product» и значением «pk» продукта, которое получаем из «field.value». По сути, мы кешируем виджет «product», в котором выбран именно этот продукт. Не всегда такое кеширование эффективно - на получение «field.value» и манипуляции с кешем (сохранение и загрузка) - тратятся ресурсы. В результате мы можем выйти в ноль эффекта или даже ухудшить производительность.

Проверяем производительность для адресов:

urls.txt

```
http://192.168.0.98/order/update/30/
http://192.168.0.98/order/update/29/
http://192.168.0.98/order/update/27/
http://192.168.0.98/order/update/25/
http://192.168.0.98/order/update/24/
http://192.168.0.98/order/update/23/
http://192.168.0.98/order/update/22/
http://192.168.0.98/order/update/21/
http://192.168.0.98/order/update/20/
http://192.168.0.98/order/update/19/
http://192.168.0.98/order/update/18/
http://192.168.0.98/order/update/15/
http://192.168.0.98/order/update/14/
http://192.168.0.98/order/update/13/
http://192.168.0.98/order/update/12/
http://192.168.0.98/order/update/10/
http://192.168.0.98/order/update/8/
http://192.168.0.98/order/update/7/
http://192.168.0.98/order/update/6/
```

Проведем тесты с кешированием и без:

```
      siege -f /home/django/geekshop/urls.txt -d0 -r19 -c125
```

В нашем случае время выполнения уменьшается на 23%: с 58,44 до 47,69 сек. Время жизни кеша установили большим (3600 сек), значит список продуктов магазина не должен изменяться часто.

Кеширование контроллеров

Поднимаемся на более высокий уровень - кеширование контроллеров. Технически оно реализуется очень просто - импортируем метод:

```
from django.views.decorators.cache import cache_page
```

И применяем его к контроллеру как декоратор:

geekshop/mainapp/views.py

```
...
@cache_page(3600)
def products(request, pk=None, page=1):
    ...
...
```

Удобнее, особенно при работе с CBV, применять метод в диспетчере URL:

geekshop/mainapp/urls.py

```
...
from django.views.decorators.cache import cache_page
...
urlpatterns = [
    ...
    re_path(r'^category/(?P<pk>\d+)/$', cache_page(3600)(mainapp.products)),
    ...
]
```

Именно такой способ рекомендуют разработчики Django. Кроме времени жизни кеша, можно передать аргумент «key_prefix» - префикс сайта, который может понадобиться при одновременном развертывании нескольких сайтов, и аргумент «cache» - используемый кеш. По умолчанию Django использует кеш «default».

Если вы реализуете кеш контроллера таким образом - обнаружите, что толку от него практически ноль. Ключ кеша формируется на основе некоторого механизма и фактически привязан только к URL адресу, которым вызывается контроллер. Значит, если кешировать страницу с каталогом - все пользователи будут видеть единственный ее экземпляр, который попал в кеш первым. А ведь у нас есть уникальные элементы - ссылка на личный кабинет пользователя в меню и корзина.

Решить проблему можно, изменив архитектуру приложения - реализуем загрузку страниц каталога через AJAX. При этом шапка страницы будет загружаться синхронно - ее контент не кешируем.

Начнем с шаблона «products_list.html» - вывод каталога перенесем в отдельный подшаблон «inc_products_list_content.html»:

geekshop/mainapp/templates/mainapp/products_list.html

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

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

   <div class="clr"></div>
{% endblock %}
```

geekshop/mainapp/templates/mainapp/includes/inc_products_list_content.html

```
{% load staticfiles %}

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

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

Создаем клон контроллера «products()»для вывода каталога через AJAX. Именно его будем потом кешировать:

geekshop/mainapp/views.py

```
...
from django.template.loader import render_to_string
from django.views.decorators.cache import cache_page
from django.http import JsonResponse
...
def products_ajax(request, pk=None, page=1):
   if request.is_ajax():
       links_menu = get_links_menu()

       if pk:
           if pk == '0':
               category = {
                   'pk': 0,
                   'name': 'все'
               }
               products = get_products_orederd_by_price()
           else:
               category = get_category(pk)
               products = get_products_in_category_orederd_by_price(pk)

           paginator = Paginator(products, 2)
           try:
```

```
            products_paginator = paginator.page(page)
        except PageNotAnInteger:
            products_paginator = paginator.page(1)
        except EmptyPage:
            products_paginator = paginator.page(paginator.num_pages)

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

        result = render_to_string(
                    'mainapp/includes/inc_products_list_content.html',
                    context=content,
                    request=request)

        return JsonResponse({'result': result})
```

Рендерим шаблон в строку и отправляем ответ в формате JSON. Дополнительно можем передать в функцию «render_to_string» объект «request» при помощи одноименного аргумента.

Прописываем в диспетчере URL:

geekshop/mainapp/urls.py

```
...
from django.views.decorators.cache import cache_page

app_name="mainapp"

urlpatterns = [
   re_path(r'^$', mainapp.products, name='index'),
   re_path(r'^category/(?P<pk>\d+)/$', mainapp.products, name='category'),
   re_path(r'^category/(?P<pk>\d+)/ajax/$',
           cache_page(3600)(mainapp.products_ajax)),
   re_path(r'^product/(?P<pk>\d+)/$', mainapp.product, name='product'),

   re_path(r'^category/(?P<pk>\d+)/page/(?P<page>\d+)/$',
           mainapp.products, name='page'),
   re_path(r'^category/(?P<pk>\d+)/page/(?P<page>\d+)/ajax/$',
           cache_page(3600)(mainapp.products_ajax)),
]
```

Вторая запись в диспетчере обрабатывает ссылки пагинатора. Время хранения кеша задали большим (3600 сек) - как и при кешировании полей выбора продуктов в заказе. Для упрощения формирования ссылок просто добавляем «/ajax/» в конце адреса.

Осталось создать файл со скриптами JS:

geekshop/static/js/main_scripts.js

```
$( document ).on( 'click', '.details a', function(event) {
   if (event.target.hasAttribute('href')) {
       var link = event.target.href + 'ajax/';
       var link_array = link.split('/');
       if (link_array[4] == 'category') {
           $.ajax({
               url: link,
               success: function (data) {
                   $('.details').html(data.result);
               },
           });

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

Получилось просто. Формируем ссылку, дописав «ajax/». Преобразуем адрес в массив по слешу и проверяем, что обращение идет именно на страницу каталога:

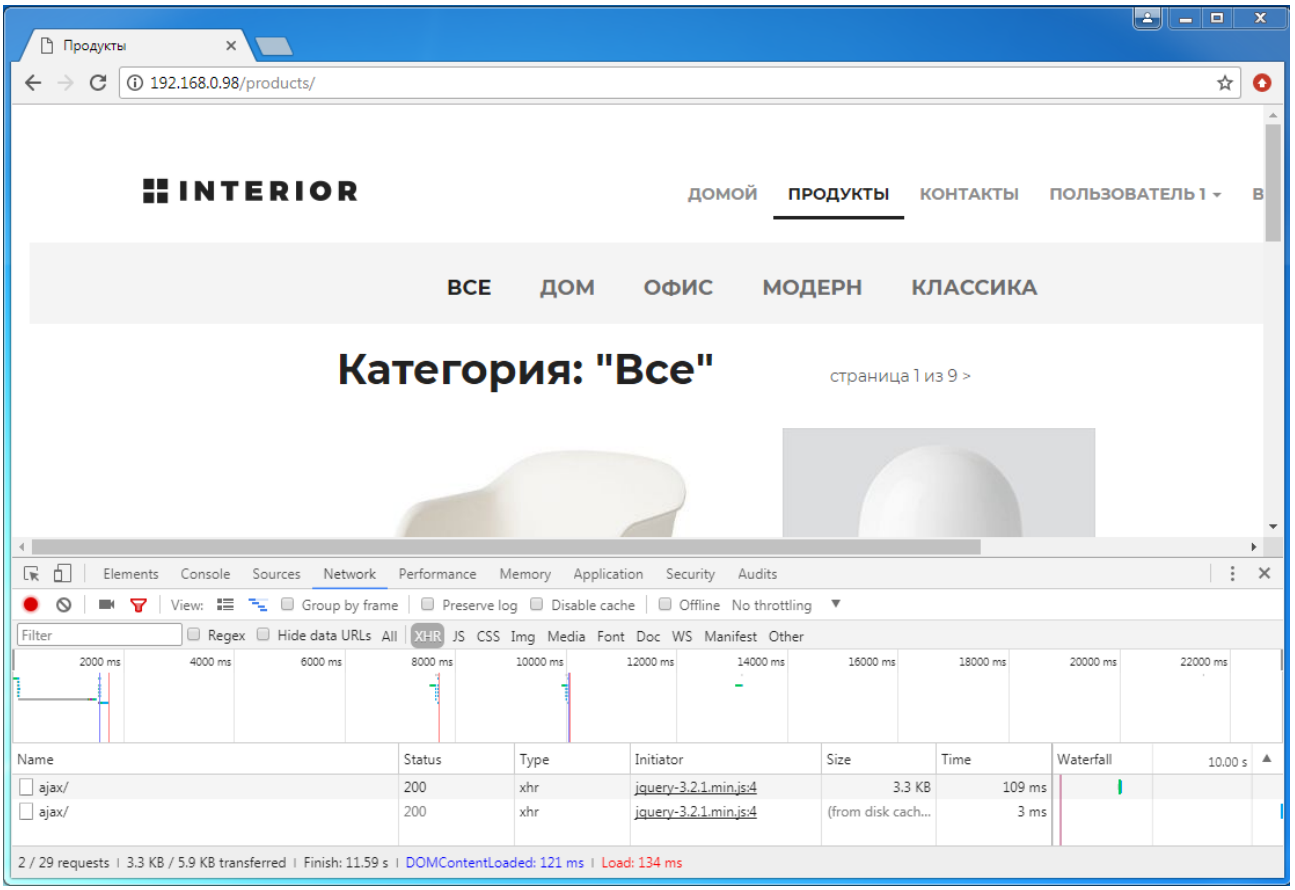
```
link_array[4] == 'category'
```

Отправляем запрос и выводим ответ при помощи метода «.html()».

Не забываем подключить скрипт в базовом шаблоне:

```
<script src="{% static 'js/main_scripts.js' %}"></script>
```

Проверим как теперь будет работать вывод каталога. Не будем касаться темы автоматизации тестирования работы web-сервера в случае с AJAX. Воспользуемся данными консоли разработчика браузера Chrome:



Видим, что загрузка раздела «Все» заняла 134 мс. Если еще раз перейти по этой ссылке - на работу AJAX затрачивается 109 мс. То есть сам по себе AJAX уже повышает производительность, ведь обновляем не всю страницу, а ее часть. Следующий вызов занимает всего 3 мс и происходит из кеша - в столбце SIZE видим «from disk cache». Так будет происходить с каждым разделом каталога, в том числе и при переходе по страницам пагинации.

Аналогично можно реализовать кеширование для других страниц, контент которых не зависит от конкретного пользователя.

Кеширование всего сайта

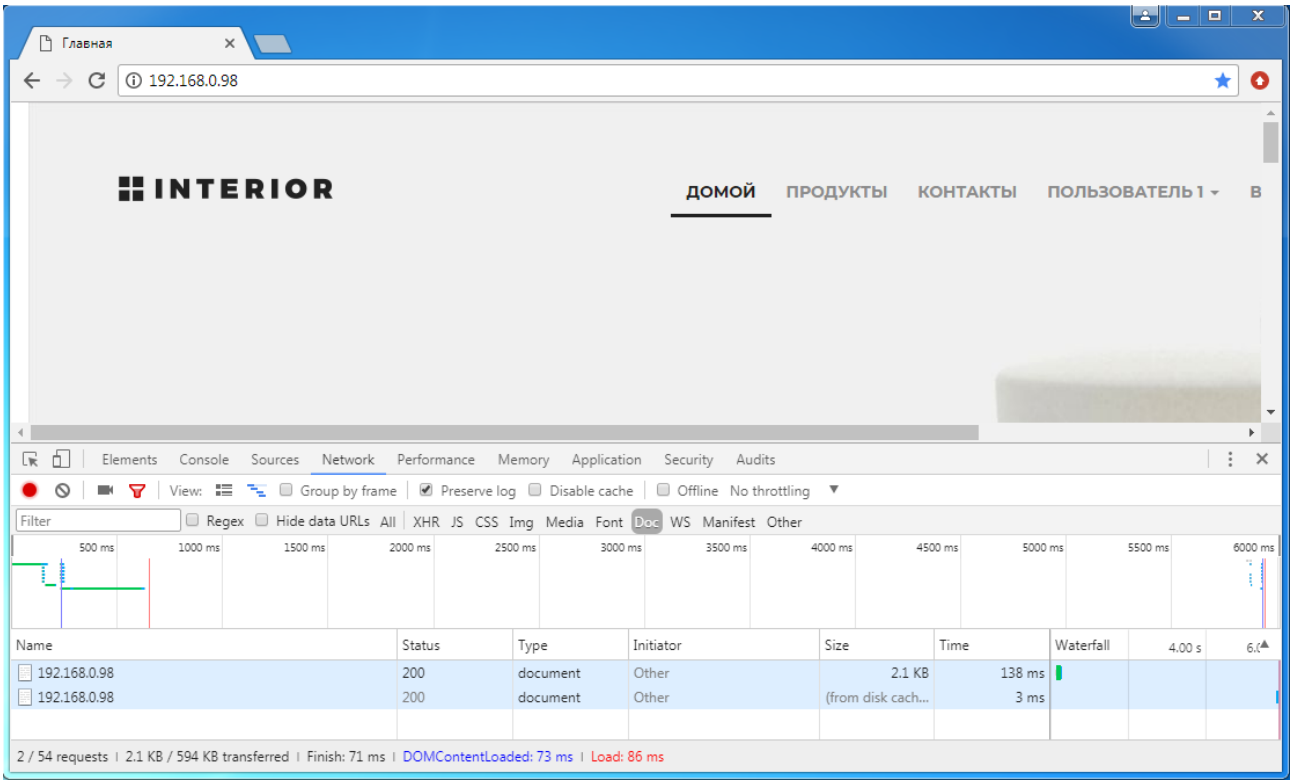
В Django можно включить кеширование для всего сайта целиком. Просто добавим две строки в файл настроек:

geekshop/geekshop/settings.py

```
...
MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.security.SecurityMiddleware',
    ...
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
]
...
```

Обязательно слой «UpdateCacheMiddleware» должен быть первым в списке, а слой «FetchFromCacheMiddleware» - последним. Первый обеспечивает обновление кешей, а последний - извлечение данных из кеша.

Проверим в браузере:



Поставили в консоли разработчика фильтр «Doc», чтобы убрать лишнюю информацию, и отметили опцию «Preserve log», чтобы сохранять данные после перезагрузки страницы. Видим, что первая загрузка страницы после рестарта сервера длилась 138 мс, а следующая уже была загружена из кеша за 3 мс.

Проведем нагрузочное тестирование для всех 49 адресов с параметрами:

```
siege -i -f /home/django/geekshop/urls.txt -d0 -r150 -c17
```

Здесь мы имитируем случайный переход каждого пользователя по всем ссылкам 3 раза (3 * 50 = 150), иначе эффект от кеша в тесте не проявится. Получили прирост производительности около 64%: с 41,74 сек до 25,38 сек.

Однако, все не так просто, есть баги. Например, при добавлении продуктов в корзину - счетчик продуктов не обновляется в строке меню. Горячее предложение, при переходе к продуктам, также не обновляется. Причина очевидна - страницы грузятся из кеша.

В Django есть декораторы для управления механизмом кеширования:

```
django.views.decorators.cache
```

Можем исключить страницу из кеша при помощи декоратора «@never_cache»:

geekshop/mainapp/views.py

```
...
from django.views.decorators.cache import never_cache
...
@never_cache
def products(request, pk=None, page=1):
    ...
...
@never_cache
def product(request, pk):
    ...
```

Обнаруженные баги мы исправили. Однако, это не все: если теперь попробуем редактировать заказы, то обнаружим, что после сохранения обновленного заказа и попытки его снова редактировать, мы видим кешированную версию. Конечно, эта ситуация маловероятна, но она иллюстрирует особенности кеширования всего сайта целиком. Необходимо тонко настраивать этот механизм. При изменении контента - обновлять кеш с помощью сигналов и т.д.

Все же, более гибкими являются методы кеширования, рассмотренные ранее. Поэтому выключим кеширование всего сайта в проекте.

Практическое задание

1. Найти в проекте повторяющиеся вызовы методов для одного экземпляра модели и применить к ним декоратор «@cached_property». Оценить, насколько уменьшилось число дублей при выполнении SQL-запросов и каков прирост производительности.
2. Применить тег «with» в одном из шаблонов. Оценить, насколько уменьшилось число дублей при выполнении SQL-запросов и каков прирост производительности.
3. Установить и настроить приложение «Memcached». Реализовать кеширование на низком уровне для функций, возвращающих редко изменяющиеся данные (продукты каталога, список категорий и т.д.). Оценить прирост производительности.
4. Реализовать кеширование в шаблоне для набора форм. Оценить эффект.
5. Реализовать работу с некоторыми пунктами меню через AJAX и кешировать соответствующие страницы. Оценить эффект от применения технологии AJAX и эффект от кеширования.
6. *Попробовать реализовать кеширование всего сайта в проекте. Оценить прирост производительности и возникающие при этом проблемы с обновлением контента.

Дополнительные материалы

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

1. Декоратор «cached_property»
2. Template tag WITH
3. Django2 cache
4. Система кеширования Django 1.9 (русский)
5. Кеширование на низком уровне
6. Кеширование «view»
7. Кеширование всего сайта
8. Декораторы управления кешированием

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация