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](#_6ksxa4eovw74)

[Встроенные механизмы кеширования](#_6utjk1tw0ntb)

[Кеширование вычисляемых полей моделей](#_1vof43w4hkky)

[Кеширование в шаблонах при помощи тега «with»](#_5a0g7djp29cm)

[Кеширование при помощи Memcached](#_8xljhrn837am)

[Установка и настройка Memcached](#_iwt3dfsk0oph)

[Низкоуровневое кеширование](#_bbvdv5hy6sdc)

[Кеширование фрагментов шаблона](#_3jli7elvki73)

[Кеширование контроллеров](#_agher56az1ri)

[Кеширование всего сайта](#_8ebmu7s5mjqn)

[Практическое задание](#_6s2b2mejcfxe)

[Дополнительные материалы](#_rfqq92u0wwsl)

[Используемая литература](#_dbkanxc9fxkf)

# 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](https://docs.djangoproject.com/en/2.0/ref/utils/" \l "module-django.utils.functional)» 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](https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#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](https://docs.djangoproject.com/en/2.0/topics/cache/) настраивается достаточно гибко:

* [кеширование всего сайта](https://docs.djangoproject.com/en/2.0/topics/cache/#the-per-site-cache);
* [кеширование контроллеров](https://docs.djangoproject.com/en/2.0/topics/cache/#the-per-view-cache);
* [кеширование фрагментов шаблона](https://docs.djangoproject.com/en/2.0/topics/cache/#template-fragment-caching);
* [низкоуровневое кеширование](https://docs.djangoproject.com/en/2.0/topics/cache/#the-low-level-cache-api).

## 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](https://docs.djangoproject.com/en/2.0/ref/settings/#std:setting-CACHES)). Если вы будете использовать несколько бекэндов для кеширования - [лучше](https://docs.djangoproject.com/en/2.0/topics/cache/#the-low-level-cache-api) 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:

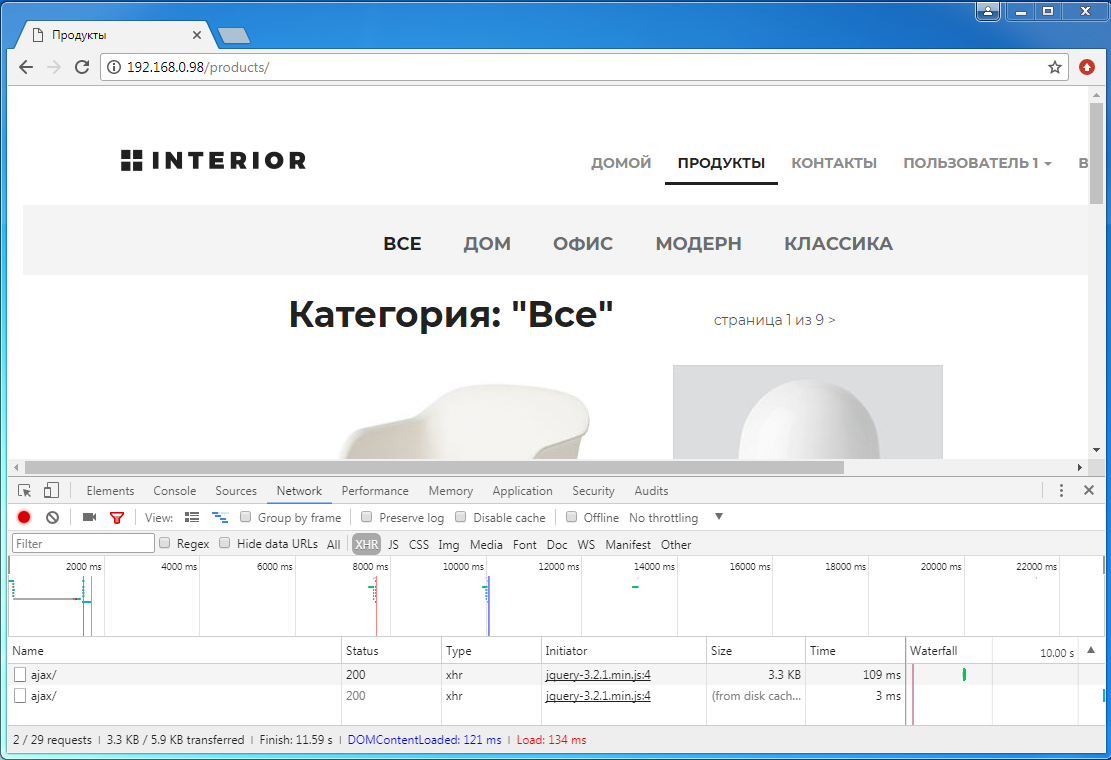
|  |
| --- |
| link\_array[4] == 'category' |

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

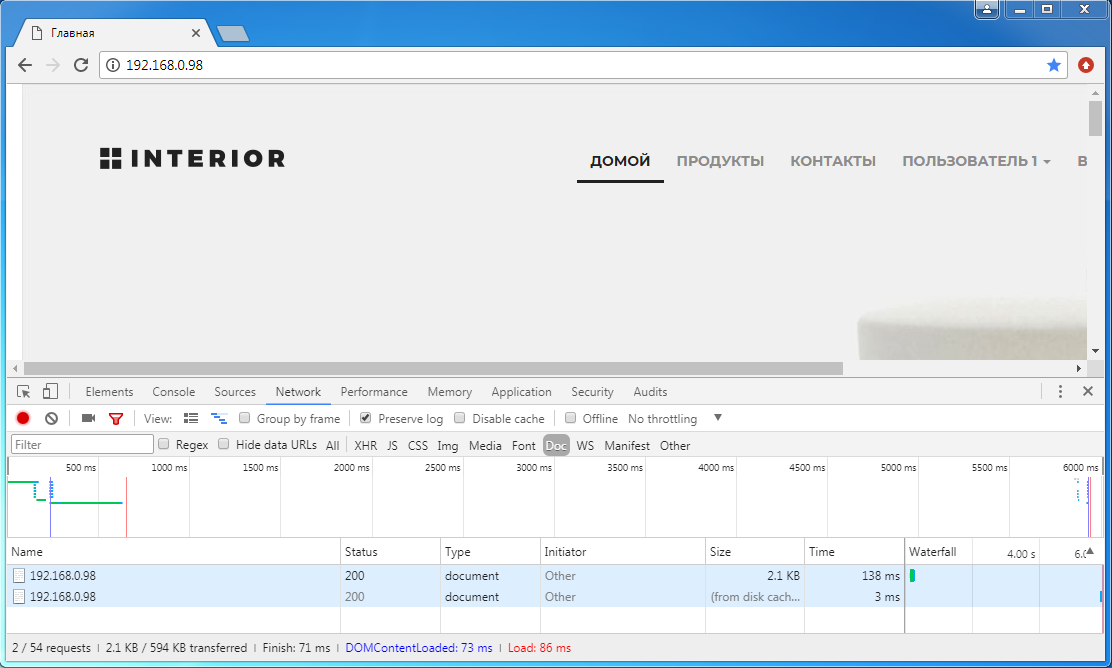
В Django можно включить кеширование для всего сайта целиком. Просто добавим две строки в файл настроек:

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](https://docs.djangoproject.com/en/2.0/topics/http/decorators/#module-django.views.decorators.cache) 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](https://docs.djangoproject.com/en/2.0/topics/http/decorators/#django.views.decorators.cache.cache_control) 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»](https://docs.djangoproject.com/en/2.0/ref/utils/#module-django.utils.functional)
2. [Template tag WITH](https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#with)
3. [Django2 cache](https://docs.djangoproject.com/en/2.0/topics/cache/)
4. [Система кеширования Django 1.9 (русский)](https://djbook.ru/rel1.9/topics/cache.html)
5. [Кеширование на низком уровне](https://docs.djangoproject.com/en/2.0/topics/cache/#the-low-level-cache-api)
6. [Кеширование «view»](https://docs.djangoproject.com/en/2.0/topics/cache/#the-per-view-cache)
7. [Кеширование всего сайта](https://docs.djangoproject.com/en/2.0/topics/cache/#the-per-site-cache)
8. [Декораторы управления кешированием](https://docs.djangoproject.com/en/2.0/topics/http/decorators/#module-django.views.decorators.cache)