Профилирование и нагрузочное тестирование проекта, оптимизация работы с базой данных

Смотрим на работу проекта изнутри. Находим медленные контроллеры. Проводим нагрузочное тестирование. Оптимизируем работу с базой данных.

[Профилирование Django проекта](#_8xljhrn837am)

[Установка и настройка «django-debug-toolbar»](#_iwt3dfsk0oph)

[Анализ данных «django-debug-toolbar»](#_fgrhtd2y7cs1)

[Приложение «django\_extensions»](#_o9dlragsf3h9)

[Тестирование производительности Django-проекта](#_bunrvs6iq4xm)

[Тестирование работоспособности](#_wznm8w9auc18)

[Нагрузочное тестирование](#_y54z3tlk941r)

[Оптимизация работы с базой данных](#_ea3p5suhzsr7)

[Контекстный процессор «basket»](#_haiydjplymr9)

[Приложение «ordersapp»](#_379aw15rz1jt)

[Добавляем индексы к атрибутам моделей](#_d65yq5dlqqyk)

[Практическое задание](#_7cj9533x7u29)

[Дополнительные материалы](#_h60hccv6bjwl)

[Используемая литература](#_bqicuolx75qt)

# 

# Профилирование Django проекта

После того, как мы развернули проект на сервере, возникает следующая важная задача - обеспечение максимально возможной производительности. Нам необходимы инструменты для профилирования, позволяющие ее измерить и найти слабые места в системе (бутылочное горло). Воспользуемся для этого инструментом «django-debug-toolbar».

## Установка и настройка «django-debug-toolbar»

Устанавливаем в виртуальное окружение:

|  |
| --- |
| pip install django-debug-toolbar |

Дополнительный модуль для профилирования загрузки шаблонов:

|  |
| --- |
| pip install django-debug-toolbar-template-profiler |

Настройки проекта:

geekshop/geekshop/settings.py

|  |
| --- |
| DEBUG = True  *# DEBUG = False*  ALLOWED\_HOSTS = ['\*'] ... INSTALLED\_APPS = [  ...  'debug\_toolbar',  'template\_profiler\_panel', ]  MIDDLEWARE = [  ...  'debug\_toolbar.middleware.DebugToolbarMiddleware', ]  if DEBUG:  def show\_toolbar(request):  return True   DEBUG\_TOOLBAR\_CONFIG = {  'SHOW\_TOOLBAR\_CALLBACK': show\_toolbar,  }   DEBUG\_TOOLBAR\_PANELS = [  'debug\_toolbar.panels.versions.VersionsPanel',  'debug\_toolbar.panels.timer.TimerPanel',  'debug\_toolbar.panels.settings.SettingsPanel',  'debug\_toolbar.panels.headers.HeadersPanel',  'debug\_toolbar.panels.request.RequestPanel',  'debug\_toolbar.panels.sql.SQLPanel',  'debug\_toolbar.panels.templates.TemplatesPanel',  'debug\_toolbar.panels.staticfiles.StaticFilesPanel',  'debug\_toolbar.panels.cache.CachePanel',  'debug\_toolbar.panels.signals.SignalsPanel',  'debug\_toolbar.panels.logging.LoggingPanel',  'debug\_toolbar.panels.redirects.RedirectsPanel',  'debug\_toolbar.panels.profiling.ProfilingPanel',  'template\_profiler\_panel.panels.template.TemplateProfilerPanel',  ] *#STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')* ... |

Снова включили режим отладки.

Добавили приложения «debug-toolbar» и «template\_profiler\_panel» в список «INSTALLED\_APPS» и слой «debug\_toolbar.middleware.DebugToolbarMiddleware» в «MIDDLEWARE».

В соответствии с [документацией](https://django-debug-toolbar.readthedocs.io/en/stable/) необходимо добавить список «INTERNAL\_IPS» с адресами, запросы с которых будут считаться локальными. Но эта настройка работает при запуске dev-сервера Django. Для работы инструментов отладки на реальном сервере создаем callback-функцию «show\_toolbar».

Также явно настраиваем список панелей «DEBUG\_TOOLBAR\_PANELS». Их названия интуитивно понятны.

Корректируем основной диспетчер URL:

geekshop/geekshop/urls.py

|  |
| --- |
| ... if settings.DEBUG:  import debug\_toolbar   urlpatterns += [re\_path(r'^\_\_debug\_\_/', include(debug\_toolbar.urls))] |

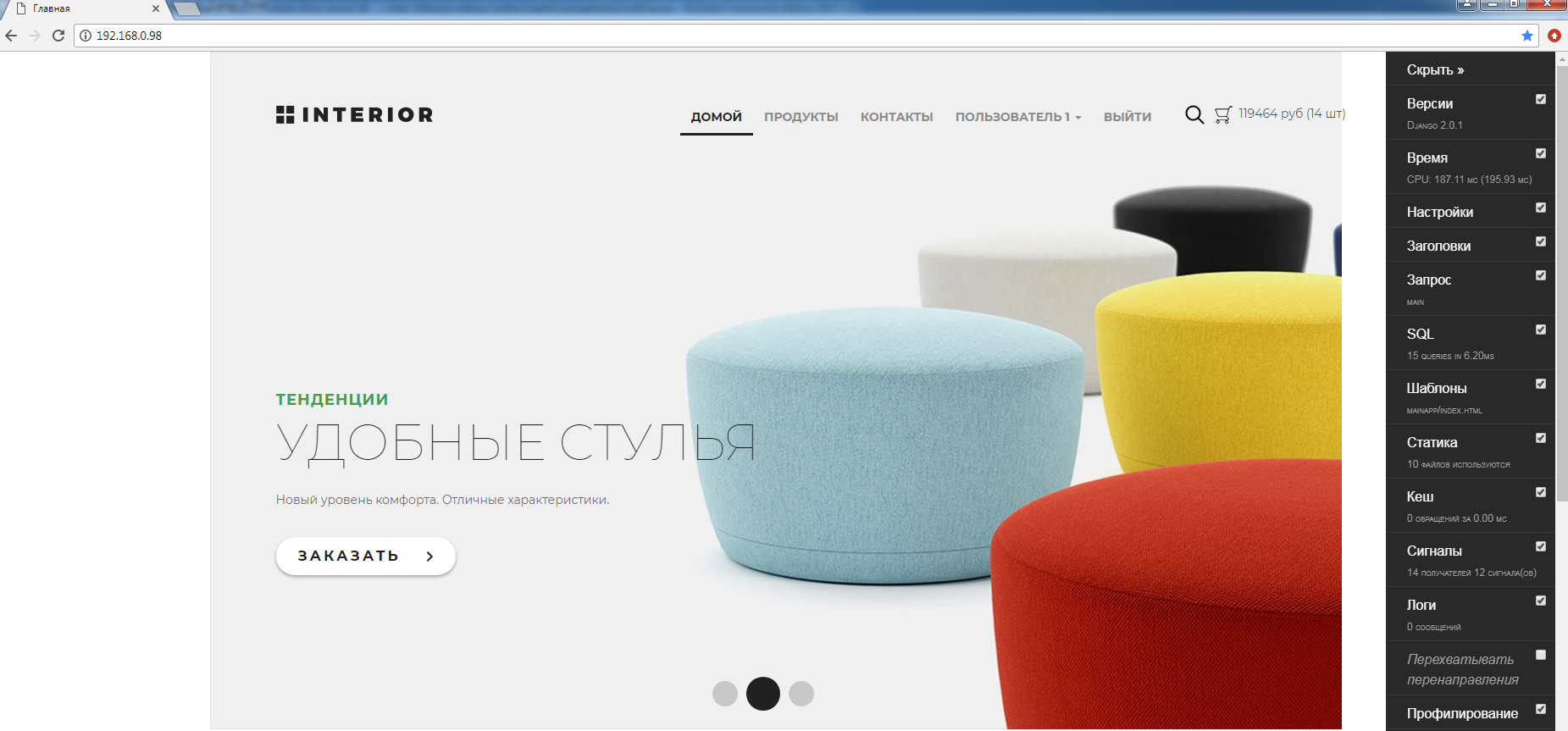
Копируем по FTP файлы на сервер. Перезагружаем службу «gunicorn»:

|  |
| --- |
| sudo systemctl restart gunicorn |

Если обновим наш сайт в браузере - увидим, что появилась отладочная информация, но без CSS стилей. Это связано с тем, что наш сервер раздает статические файлы только из папки «static», а для приложения «debug-toolbar» они находятся в папке, где оно установлено (в нашем случае это папка с виртуальным окружением). Для решения этой проблемы раскомментируем в файле настроек константу «STATIC\_ROOT» и воспользуемся инструментом Django для сбора статических файлов:

|  |
| --- |
| python manage.py collectstatic |

После его запуска в папке «static» появится папка «debug\_toolbar» и, возможно, еще папки с именами других сторонних приложений. Теперь константу «STATIC\_ROOT» необходимо снова закомментировать. После перезапуска службы «gunicorn» и обновления страницы, должны в правой части экрана увидеть панель отладки:



Если сайт не открывается - ищем ошибки в файле настроек. Проверяем статус службы «gunicorn»:

|  |
| --- |
| sudo systemctl status gunicorn |

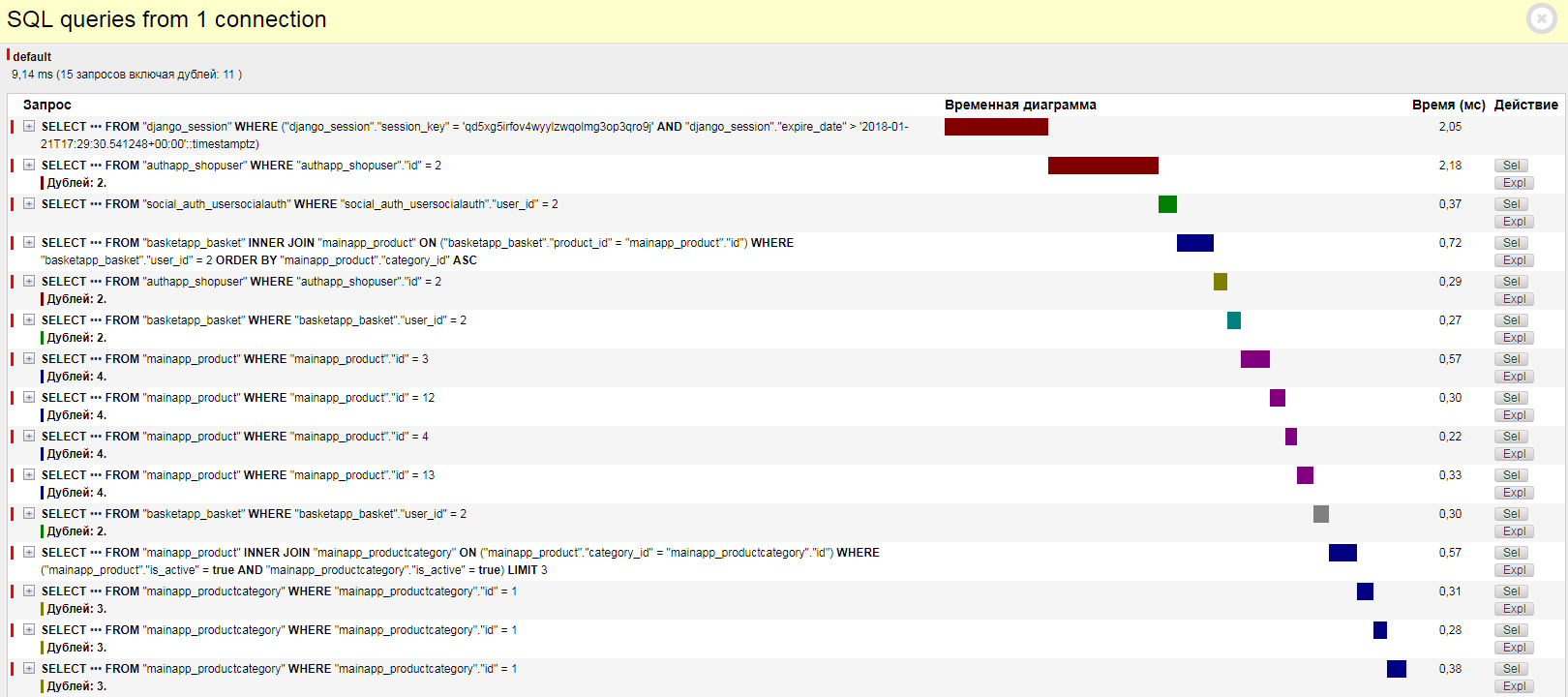
**Внимание:** при обновлении контента сайта по FTP не забывайте устанавливать параметры доступа «755», иначе могут быть недоступны статические файлы или не работать весь сайт.

## Анализ данных «django-debug-toolbar»

Нам необходимо собрать информацию о производительности сайта. Помните, что Django - это фреймворк, и, следовательно, обладает избыточностью.

На вкладке «Время» отображается время загрузки страницы, которое складывается из продолжительности работы процессора (CPU) и накладных расходов в виде переключения контекста и прочих. Запишите значения для разных страниц сайта. Таким образом мы можем узнать, какие из контроллеров работают медленнее всего.

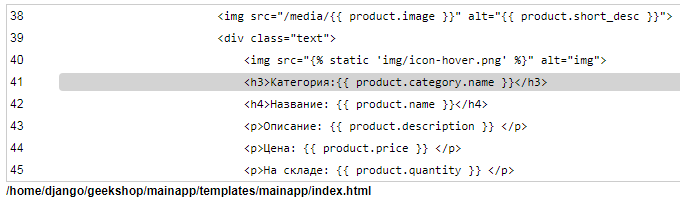
Для оптимизации работы с БД имеют большое значение данные вкладки «SQL». Зафиксируйте количество запросов и их дубликатов на страницах сайта. Обратите внимание на самые медленные запросы:



В данной ситуации видим, что при загрузке главной страницы сайта, дольше всего выполнялся запрос данных о сессии. Также относительно длительным был следующий запрос аутентификации пользователя. Затем Django пытается найти пользователя в таблице приложения «social\_django» - работает его контекстный процессор. Видим запрос контекстного процессора корзины, заканчивающийся «ORDER BY "mainapp\_product"."category\_id" ASC». Запрос продуктов из контроллера «main(request)» можно распознать по ограничению «LIMIT 3»:

|  |
| --- |
| Product.objects.filter(is\_active=True, category\_\_is\_active=True)[:3] |

Если нажать слева на символ '+', можно посмотреть детальную информацию о контексте вызова запроса. Это позволяет нам обнаружить дубликаты запросов при выводе продуктов в шаблоне:



Каждый раз при обращении к связанной модели в шаблоне «category» Django делает запрос. Это неэффективно. Правильней загрузить данные об объектах категорий продуктов, связанных через ForeignKey, вместе с данными о самих продуктах. Для этого в Django существует метод [.select-related()](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#select-related) объекта QuerySet:

geekshop/mainapp/views.py

|  |
| --- |
| ... def main(request):  title = 'главная'   products = Product.objects.\  filter(is\_active=True, category\_\_is\_active=True).\  **select\_related(**'category'**)**[:3]   content = {  'title': title,  'products': products,  }   return render(request, 'mainapp/index.html', content) ... |

Теперь число дубликатов запросов должно уменьшиться на 3 - убедитесь в этом.

Для того, чтобы понять как это работает, можно обратиться к более низкому уровню (по сравнению с ORM) языку запросов SQL. Посмотрим через атрибут «query» объекта «QuerySet» текст запроса без метода .select\_related(«category»):

|  |
| --- |
| print(products.query) |

|  |
| --- |
| SELECT  "mainapp\_product"."id", "mainapp\_product"."category\_id", "mainapp\_product"."name", "mainapp\_product"."image", "mainapp\_product"."short\_desc", "mainapp\_product"."description", "mainapp\_product"."price", "mainapp\_product"."quantity", "mainapp\_product"."is\_active"  FROM  "mainapp\_product"  INNER JOIN "mainapp\_productcategory" ON ("mainapp\_product"."category\_id" = "mainapp\_productcategory"."id")  WHERE  ("mainapp\_product"."is\_active" = True AND "mainapp\_productcategory"."is\_active" = True) LIMIT 3 |

В этом запросе получаем только данные о продуктах. Следовательно, каждый раз при выводе имени категории продукта будет новый запрос. У нас 3 продукта - получаем три дубликата.

Текст запроса с методом .select\_related(«category»):

|  |
| --- |
| SELECT  "mainapp\_product"."id", "mainapp\_product"."category\_id", "mainapp\_product"."name", "mainapp\_product"."image", "mainapp\_product"."short\_desc", "mainapp\_product"."description", "mainapp\_product"."price", "mainapp\_product"."quantity", "mainapp\_product"."is\_active",  "mainapp\_productcategory"**.**"id"**,** "mainapp\_productcategory"**.**"name"**,** "mainapp\_productcategory"**.**"description"**,** "mainapp\_productcategory"**.**"is\_active" FROM  "mainapp\_product"  INNER JOIN "mainapp\_productcategory" ON ("mainapp\_product"."category\_id" = "mainapp\_productcategory"."id")  WHERE ("mainapp\_product"."is\_active" = True AND "mainapp\_productcategory"."is\_active" = True) LIMIT 3 |

Появились две строки в «SELECT» - сразу получаем данные о категориях, поэтому дубликатов запросов нет. На низком уровне .select\_related() работает через «SELECT».

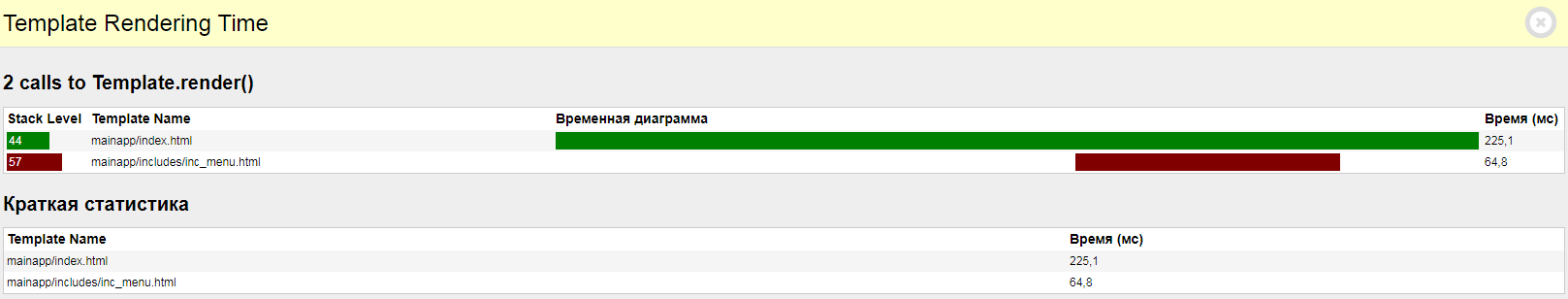
Лучше явно указывать имена подгружаемых полей как аргументы при вызове .select\_related(), иначе Django будет включать в запрос все поля модели с непустым значением «ForeignKey».

Для проведения дальнейшей оптимизации работы с БД, запишем количество запросов и дублей для страниц нашего ресурса (пользователь «user1» и в корзине 16 товаров):

|  |  |  |
| --- | --- | --- |
| **Адрес** | **Число запросов** | **Число дублей** |
| '/' | 14 | 10 |
| '/products/' | 20 | 14 |
| '/products/category/1/' | 17 | 10 |
| '/basket/' | 25 | 22 |
| '/order/' | 4 | 0 |
| '/order/update/24/' | 82 | 77 |

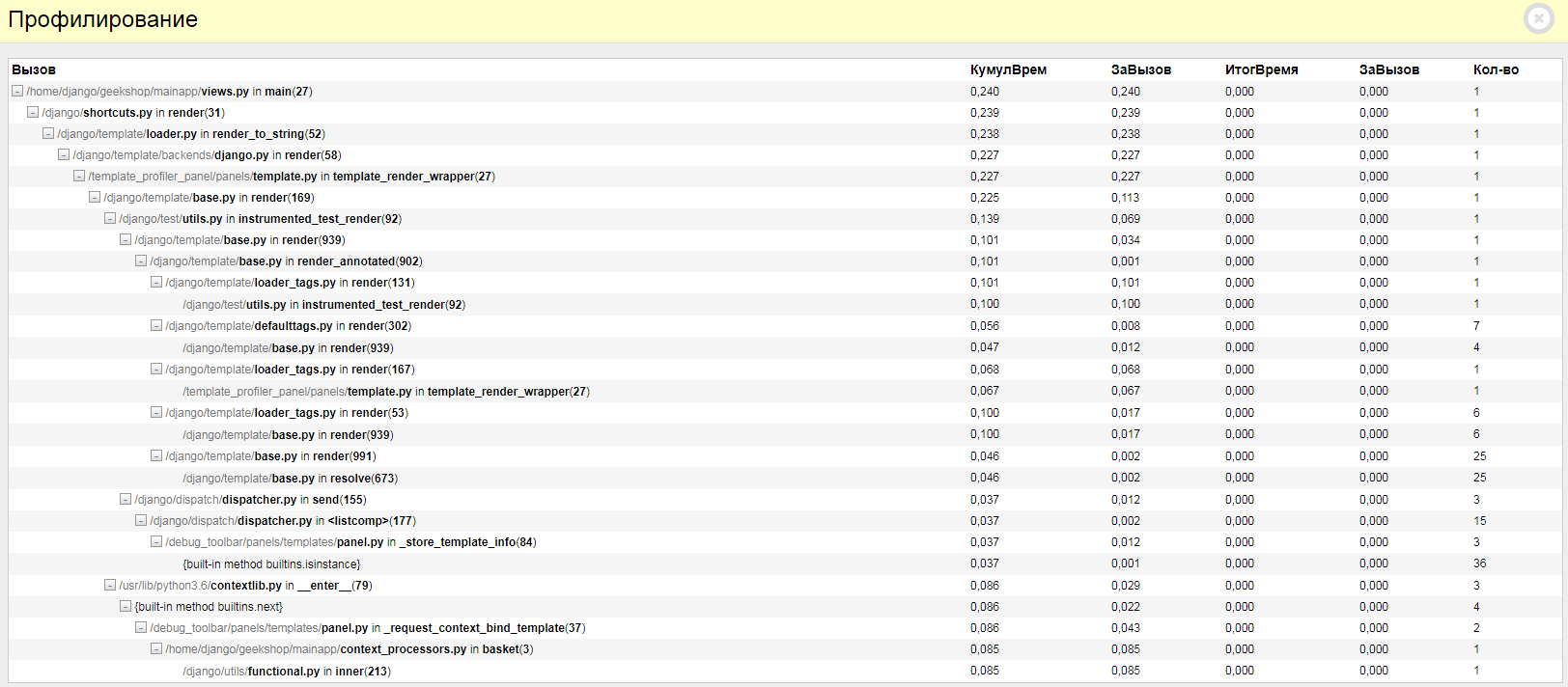
Из таблицы видно, что начинать оптимизацию необходимо с контроллера редактирования заказа, потом будем работать с корзиной и списком продуктов.

Продолжим анализ информации, полученной с помощью **«**django-debug-toolbar**»**. На самой нижней вкладке **«**Template Profiler**»** отображается время рендеринга шаблонов и подшаблонов:



Видно, что подшаблон «inc\_menu.html» начал рендериться и был полностью обработан в процессе рендеринга основного шаблона «index.html». В будущем можем использовать эти данные для кеширования шаблонов.

Дальше переходим ко вкладке «Профилирование», позволяющей увидеть работу Django-приложения на уровне Python:



Здесь видна статистика вызовов функций: количество, накопленное время и другая информация.

Итак, теперь у нас в руках есть мощный инструмент для разностороннего анализа работы проекта - «django-debug-toolbar».

## Приложение «django\_extensions»

Для анализа Django проекта установим еще одно приложение - «[django\_extensions](https://pypi.python.org/pypi/django-extensions/1.9.9)»:

|  |
| --- |
| pip install django-extensions |

Традиционно добавим строку в список «INSTALLED\_APPS»:

|  |
| --- |
| 'django\_extensions' |

Теперь можем одной командой собрать данные из всех диспетчеров URL проекта:

|  |
| --- |
| python manage.py show\_urls > geekshop\_urls.txt |

Или выполнить валидацию шаблонов проекта:

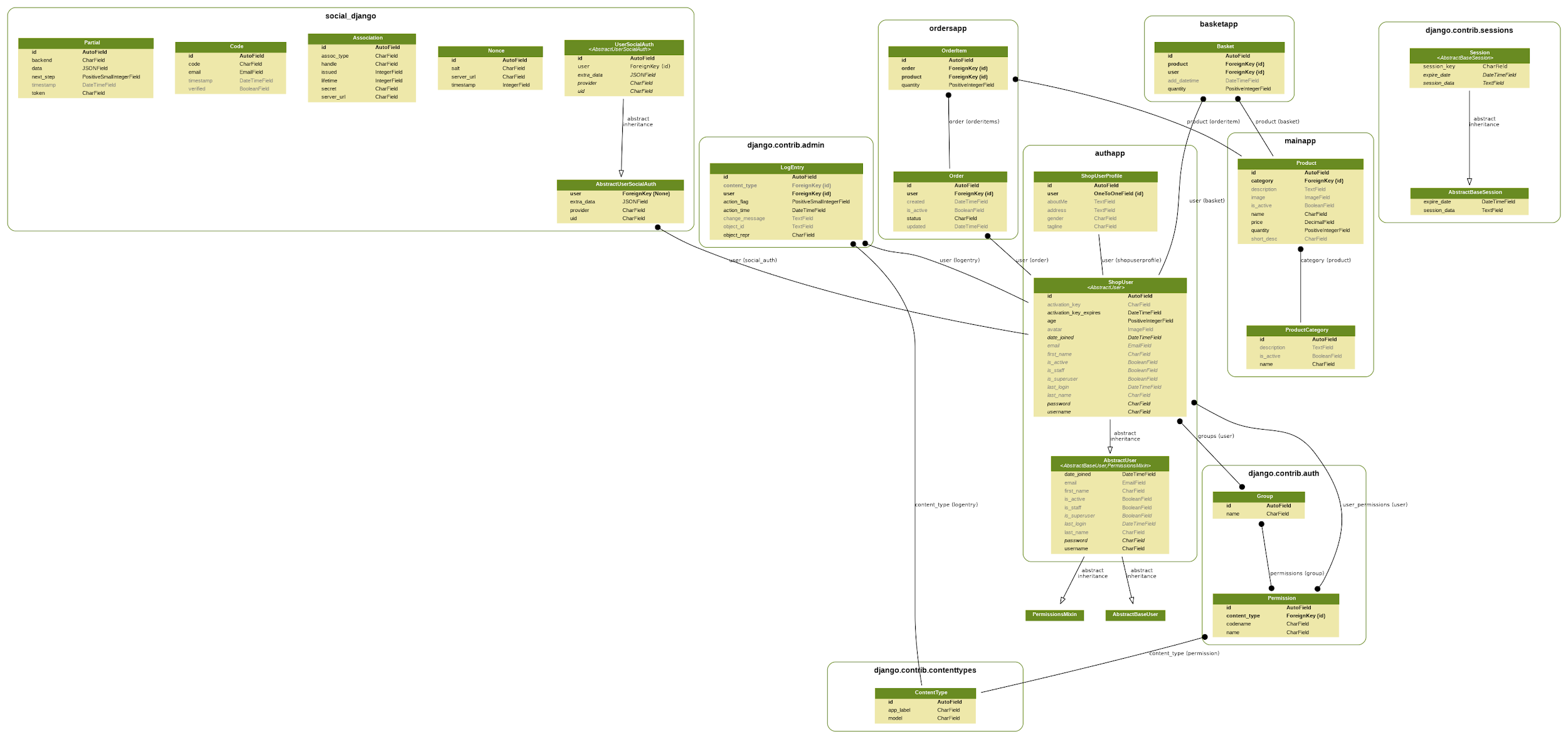
|  |
| --- |
| python manage.py validate\_templates |

Установим еще одно приложение:

|  |
| --- |
| pip install pydotplus |

Это позволит [визуализировать](https://django-extensions.readthedocs.io/en/latest/graph_models.html) структуру моделей проекта:

|  |
| --- |
| python manage.py graph\_models -a -g -o geekshop\_visualized.png |



Для более глубокого анализа работы проекта можно запустить [сервер](https://django-extensions.readthedocs.io/en/latest/runprofileserver.html) с записью результатов профилирования в файлы с расширением .prof:

|  |
| --- |
| python manage.py runprofileserver *--kcachegrind*   *--prof-path=tmp/my-profile-data 0.0.0.0:8000* |

Это расширенная версия dev-сервера Django. Перед запуском необходимо создать папку для хранения файлов, например:

|  |
| --- |
| mkdir tmp/my-profile-data2 |

В дальнейшем их можно визуализировать, например, при помощи инструмента «[kcachegrind](http://kcachegrind.sourceforge.net/html/Home.html)» (работает в \*nix системах).

**Внимание:** не забывайте добавить новые приложения в файл «requirements.in».

# Тестирование производительности Django-проекта

Теперь пришло время оценки производительности в условиях, максимально приближенных к реальной эксплуатации. Следует понимать, что результаты тестирования носят оценочный характер и зависят от большого числа факторов. Важны не абсолютные значения результатов, а их динамика в ходе оптимизации сайта.

Существует большое количество инструментов для тестирования web-серверов: [apache2-utils](https://packages.ubuntu.com/artful/net/apache2-utils), [siege](https://github.com/JoeDog/siege) и другие. Мы будем использовать утилиту «siege», создающую интенсивную нагрузку на сервер:

|  |
| --- |
| sudo apt install siege |

## Тестирование работоспособности

Первый тест будет простой - проверим статус ответа основных контроллеров.

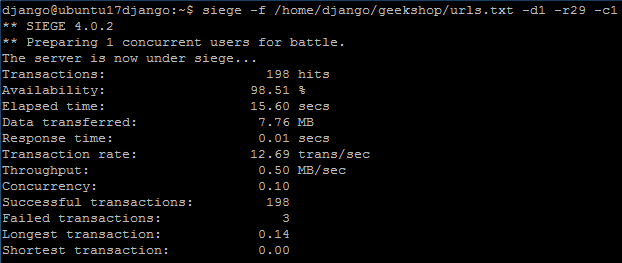
Отключаем в проекте режим отладки и создаем в корне проекта файл с их адресами (можно воспользоваться созданным ранее файлом «geekshop\_urls.txt»):

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/ http://192.168.0.98/basket/ http://192.168.0.98/order/ http://192.168.0.98/auth/edit/ 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/ |

Выполним тест-команду:

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d1 -r29 -c1 |



Интерпретация ключей:

* «-f» - используем файл со списком адресов;
* «-d1» - задержка между запросами от 0 до 1 секунды;
* «-r28» - каждый пользователь посылает 29 запросов;
* «-c1» - имитируем работу одного пользователя.

Видим, что три запроса были неудачными (Failed transactions). Скорректируем файл с адресами:

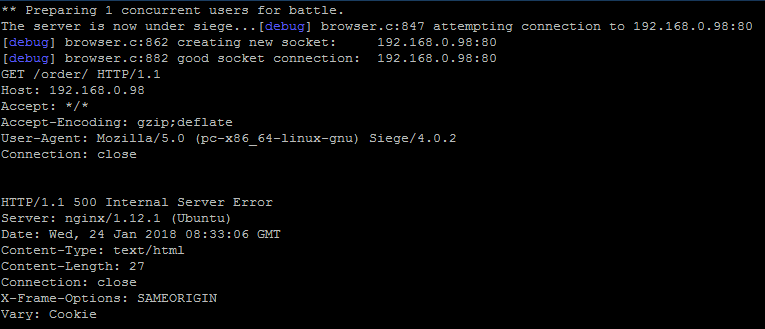
urls.txt

|  |
| --- |
| http://192.168.0.98/order/ http://192.168.0.98/auth/edit/ http://192.168.0.98/contact/ |

Повторим тест с параметром «--debug»:

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d1 -r3 -c1 --debug |

Фрагмент ответа:



В режиме «DEBUG» увидели бы отладочные данные. Раз он выключен - «ошибка 500». Рекомендуем снова включить «DEBUG» и проверить все три адреса из списка «urls.txt» в браузере, когда пользователь не залогинен.

Исправим ошибки в проекте. Для контроллера «/order/» добавим декоратор:

geekshop/ordersapp/views.py

|  |
| --- |
| ... **from django.utils.decorators import method\_decorator from django.contrib.auth.decorators import login\_required**   class OrderList(ListView):  model = Order   def get\_queryset(self):  return Order.objects.filter(user=self.request.user)   **@method\_decorator(login\_required())  def dispatch(self, \*args, \*\*kwargs):  return super(ListView, self).dispatch(\*args, \*\*kwargs)** |

Таким же способом задекорируем контроллеры для создания, детального просмотра и редактирования заказа. Добавим обычный декоратор @login\_required для контроллера «edit()» в приложении «authapp».

Ошибки в контроллере «contact()» приложения «mainapp» при запуске dev-сервера в Windows не было, но при запуске production-сервера она появилась. Уточним функцию «load\_from\_json()»:

geekshop/mainapp/views.py

|  |
| --- |
| **... def load\_from\_json(file\_name):  with open(os.path.join(JSON\_PATH, file\_name +** '.**json'),** 'r'**,\**  **errors=**'ignore'**) as infile:  return json.load(infile) ...** |

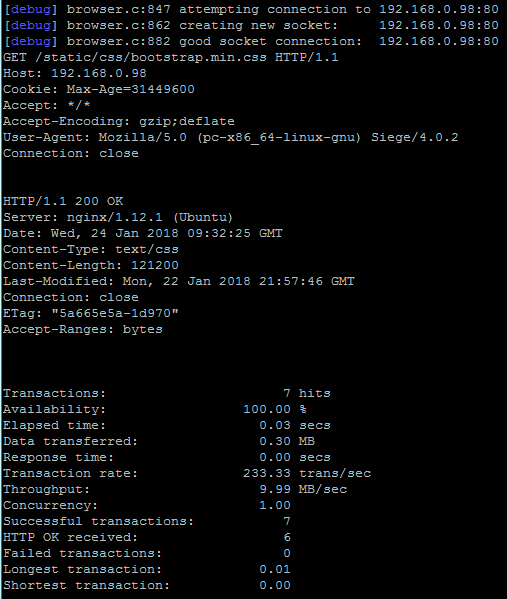
Добавим к изначальному содержимому файла «urls.txt» адреса для просмотра подробной информации о заказах «/order/update/» и страницу регистрации пользователя «/auth/register/». Итого получим 49 элементов. Повторим тестирование:

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d1 -r49 -c1 |

Выполнено 358 запросов, ошибок нет. Однако, есть другая проблема - в большинстве случаев сервер нам возвращает страницу входа в систему вместо реального контента. Можете увидеть это запуская тест по отдельным адресам с параметром «--debug». Например, для запроса:

|  |
| --- |
| siege http://192.168.0.98/order/read/24/ -d0 -r1 -c1 --debug |

Получим 7 ответов, последний будет таким:



Видим, что Cookie до сих пор пустые - значит пользователь в систему не вошел. Если проанализировать более пристально ответы, то увидите, что среди них нет ни одного изображения товара, а их должно быть в этом заказе три.

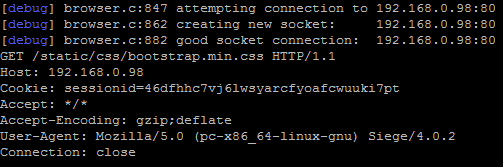
Утилита «siege» позволяет передавать POST-данные в запросах:

urls.txt

|  |
| --- |
| http://192.168.0.98/auth/login/ POST username=user1&password=geekbrains http://192.168.0.98/order/read/24/ |

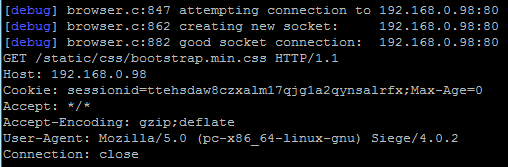
Если сейчас запустить тест - увидим, что логина так и не произошло. Причина - Django не принимает POST-запросы без корректного значения «CSRF», так как по умолчанию работает слой защиты от подделки запроса «django.middleware.csrf.CsrfViewMiddleware». На время тестирования его необходимо закомментировать в списке «MIDDLEWARE» файла настроек и *перезапустить* сервер «gunicorn». Теперь все хорошо:

siege -f /home/django/geekshop/urls.txt -d0 -r2 -c1 --debug



Видим, что в Cookie появился ключ «sessionid», значит пользователь залогинился. Если добавить третий адрес «http://192.168.0.98/auth/logout/» и выполнить тест, увидим:

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d0 -r3 -c1 --debug |



Значение «Max-Age=0» говорит, что ключ недействителен и пользователь уже не в системе.

Мы в ходе тестирования будем имитировать режим интернета (ключ '-i'), когда пользователи переходят по адресам в случайном порядке. Поэтому логин должен происходить автоматически при переходе по любому адресу. Можно переписать адреса в виде:

|  |
| --- |
| http://192.168.0.98/auth/login/?next=/order/read/24/ POST username=user1&password=geekbrains |

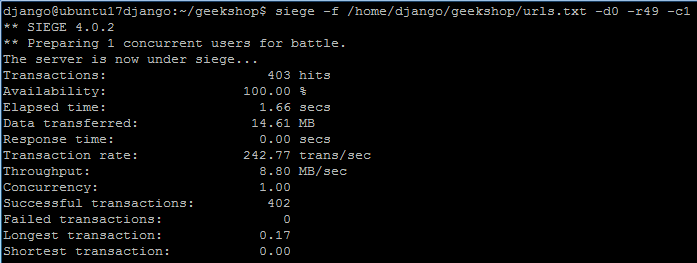
Вместо этого добавим в файл настроек «/home/django/.siege/siege.conf» строку:

|  |
| --- |
| login-url = http://192.168.0.98/auth/login/ POST username=user1&password=geekbrains |

Таких строк может быть несколько для имитации работы множества пользователей. Изучите внимательно файл настроек - увидите много новых возможностей для тестирования.

Теперь вернем в файл «urls.txt» все 49 адресов и выполним тест:

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d0 -r49 -c1 |



Ошибок нет - функциональное тестирование пройдено.

Если вернуть изначальные 29 адресов и провести тест, то увидим, что число транзакций увеличилось с 198 до 226 - ведь теперь грузится реальный контент, а не страницы логина. Рекомендуем поработать с тестами более подробно через [less](https://ru.wikipedia.org/wiki/Less):

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d0 -r49 -c1 --debug | less |

## Нагрузочное тестирование

Сначала выполним тестирование конкретных контроллеров, записывая по одному адресу в файл «urls.txt» и выполняя тест:

|  |
| --- |
| siege -f /home/django/geekshop/urls.txt -d0 -r25 -c50 |

В результате заполним таблицу (пользователь «user1», в корзине 16 товаров):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Адрес** | **Переходов** | **Время теста, с** | **Транзакций в секунду** | **Время отклика, с** |
| / | 13850 | 16,30 | 849,69 | 0,06 |
| /products/ | 15017 | 18,91 | 794,13 | 0,06 |
| /products/category/1/ | 11550 | 17,21 | 671,12 | 0,07 |
| /basket/ | 15350 | 20,04 | 765,97 | 0,06 |
| /order/ | 9150 | 13,68 | 668,86 | 0,07 |
| /order/update/24/ | 9150 | 49,20 | 185,98 | 0,26 |
| /order/read/24/ | 12900 | 15,27 | 844,79 | 0,06 |

Ожидаемо самым медленным оказался контроллер редактирования заказа («/order/update/24/»). Именно с него будем начинать оптимизацию. Доступность сервера в тестах была 100%, самая долгая транзакция порядка 3 секунд. Если увеличить число пользователей до 140 и уменьшить число переходов каждого из них до 5 - начнутся отказы. Обязательно узнайте число пользователей, при котором начинаются отказы для своей системы.

Дальше проведем тестирование в условиях, максимально близко имитирующих реальную работу сервера - включим режим интернета (ключ «-i») - адреса переходов будут выбираться случайным образом. Вернем в файл «urls.txt» все 49 адресов. Результаты тестов для разных значений параметров запишем в таблицу:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Параметр** | **-r50 -c50** | **-r25 -c100** | **-r17 -c150** | **-r12 -c200** |
| Число переходов | 20829 | 21013 | 19061 | 13296 |
| Доступность сервера | 100% | 100% | 98,19% | 93,85% |
| Время теста, с | 52,12 | 51,95 | 49,01 | 36,95 |
| Время отклика, с | 0,12 | 0,24 | 0,33 | 0,35 |
| Запросов в секунду | 399,64 | 404,49 | 388,92 | 359,84 |
| Пропускная способность,  МБ/сек | 14,34 | 14,40 | 13,59 | 12,52 |
| Согласованность | 49,46 | 98,24 | 126,46 | 126,08 |
| Удачных транзакций | 20789 | 20956 | 19029 | 13281 |
| Неудачных транзакций | 0 | 0 | 351 | 872 |
| Самая долгая транзакция, с | 3,04 | 5,84 | 7,88 | 8,26 |

Из тестов видно, что наш ресурс может без отказов ответить примерно 126 различным пользователям. При увеличении числа пользователей ожидаемо возрастает время отклика и самая большая длительность транзакции. Пришло время заняться оптимизацией.

# Оптимизация работы с базой данных

В идеале после каждого шага необходимо повторять тесты и оценивать эффективность.

## Контекстный процессор «basket»

Добавим метод «.select\_related()»:

|  |
| --- |
| basket = request.user.basket.select\_related() |

Включаем режим «DEBUG» и смотрим как изменилось число запросов на страницах:

|  |  |  |
| --- | --- | --- |
| **Адрес** | **Число запросов (изменение)** | **Число дублей (изменение)** |
| '/' | 13 (-1) | 8 (-2) |
| '/products/' | 19 (-1) | 12 (-2) |
| '/products/category/1/' | 16 (-1) | 8 (-2) |
| '/basket/' | 12 (-13) | 8 (-14) |
| '/order/' | 4 (0) | 0 (0) |
| '/order/update/24/' | 82 (0) | 77 (0) |

## Приложение «ordersapp»

Добавим метод .select\_related() при загрузке продуктов в форме элемента заказа (класс «OrderItemForm»):

|  |
| --- |
| self.fields['product'].queryset = Product.get\_items().select\_related() |

Число запросов на странице «/order/update/24/» сразу уменьшилось с 82 до 14 (9 дублей)!

Если сейчас проведем тест:

|  |
| --- |
| siege -i -f /home/django/geekshop/urls.txt -d0 -r17 -c150 |

Обнаружим, что время отклика уменьшилось на 21% с 0,33 сек до 0,26 сек.

Добавим метод «.select\_related()» в контроллере редактирования заказа:

geekshop/ordersapp/views.py

|  |
| --- |
| ... class OrderItemsUpdate(UpdateView):  ...    def get\_context\_data(self, \*\*kwargs):  ...  if self.request.POST:  ...  else:  queryset = self.object.orderitems.select\_related()  formset = OrderFormSet(instance=self.object, queryset=queryset)  ... |

Теперь число запросов на странице «/order/update/24/» равно 11 (6 дублей).

Аналогичные действия необходимо проделать в остальных приложениях - ищем запросы, в которых получаем объекты, связанные с другими объектами через внешний ключ или через отношение один-к-одному и добавляем метод .select\_related(). Обязательно проводите тесты после корректировок. Усложнение запросов может вызвать уменьшение производительности, несмотря на уменьшение их количества.

Если сейчас провести сравнительное тестирование только для страницы «/order/update/24/» - обнаружим прирост производительности порядка 70%:

* время отклика уменьшилось с 0,26 с до 0,15 с;
* время выполнения теста уменьшилось с 49,2 с до 28,72 с;
* число транзакций в секунду возросло со 185,98 до 318,59.

**Внимание:** в Django есть еще один полезный для оптимизации метод «[.prefetch-related()](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#prefetch-related)». Он работает на уровне Python, а не на уровне запросов к БД. *Обязательно* познакомьтесь с ним.

## Добавляем индексы к атрибутам моделей

Кроме количества запросов на производительность проекта большое влияние оказывает скорость их выполнения. Самый эффективный способ ее повышение - добавление индексов. Нужно понимать, что этот прирост происходит за счет увеличения объема базы. Выбор полей для создания индексов - сложная задача. Мы просто покажем эффективность этого метода. Следует помнить, что индексы автоматически добавляются Django для всех полей «[ForeignKey](https://docs.djangoproject.com/en/2.0/ref/models/fields/" \l "django.db.models.ForeignKey)».

Чтобы поле индексировалось просто добавляем аргумент «db\_index=True»:

|  |
| --- |
| is\_active = models.BooleanField(db\_index=True, default=True) |

Создавать индексы необходимо для всех полей, участвующих в запросах.

|  |  |
| --- | --- |
| **Модель** | **Индексируемые поля** |
| ProductCategory | is\_active |
| Product | is\_active |
| Order | is\_active |

Так как на данном этапе база данных содержит небольшое количество записей - эффект от индексирования будет незаметен. Но в будущем он будет существенным.

# Практическое задание

1. Установить приложение «django-debug-toolbar». Оценить время загрузки страниц. Найти самые медленные контроллеры. Заполнить таблицу с количеством запросов и дубликатов на страницах проекта.
2. Визуализировать структуру моделей проекта при помощи «django\_extensions», создать файл «geekshop\_urls.txt» с URL адресами проекта.
3. Установить утилиту «siege» и провести функциональное тестирование. Зафиксировать результаты в текстовом файле (какие контроллеры работали с ошибками).
4. Провести нагрузочное тестирование отдельных страниц и записать результаты в таблицу.
5. Провести тестирование в режиме интернета. Записать данные в таблицу. Определить условия, при которых начинаются отказы.
6. Провести оптимизацию работы с БД в проекте. Оценить эффект.

# Дополнительные материалы

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

1. [Документация по «django-debug-toolbar»](https://django-debug-toolbar.readthedocs.io/en/stable/)
2. [Настройка панелей «django-debug-toolbar»](http://django-debug-toolbar.readthedocs.io/en/stable/configuration.html)
3. [Метод «QuerySet .prefetch-related()»](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#prefetch-related)
4. [Документация по «django-extensions»](https://django-extensions.readthedocs.io/en/latest/)
5. [Руководство по «kcachegrind»](https://docs.kde.org/stable5/en/kdesdk/kcachegrind/index.html)
6. [Метод «.select-related()»](https://docs.djangoproject.com/en/2.0/ref/models/querysets/#select-related)
7. [Инструмент «apache2-utils»](https://httpd.apache.org/docs/2.4/programs/ab.html)
8. [Утилита «siege» (habrahabr)](https://habrahabr.ru/post/65128/)
9. [Оптимизация работы с БД (оригинал)](https://docs.djangoproject.com/en/2.0/topics/db/optimization/)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация](https://docs.djangoproject.com/en/2.0/)