

Классы представлений, регистрация, оптимизация

Классы представлений: ListView, DetailView, CreateView

Основы ORM Django за час

Mixins - убираем дублирование кода

Постраничная навигация (пагинация)

Регистрация пользователей на сайте

Делаем авторизацию пользователей на сайте

Оптимизация сайта с Django Debug Toolbar

Включаем кэширование данных

Использование капчи captcha

Тонкая настройка админ панели

Поделиться

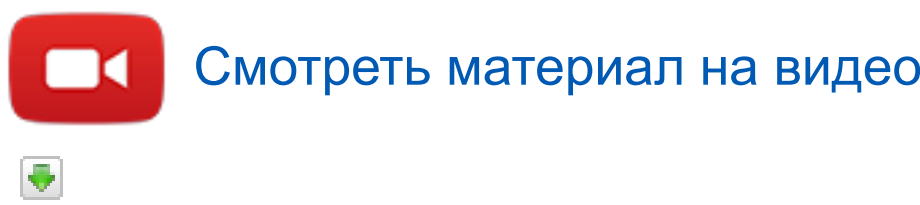


Наш канал



Главная → Django → Классы представлений, регистрация, оптимизация

Оптимизация сайта с Django Debug Toolbar



Архив проекта: [lesson-21-coolsite.zip](#)

К этому моменту мы с вами создали небольшой, но функциональный сайт с отображением статей, регистрацией и авторизацией пользователей. Но одну и ту же задачу программно можно реализовать по-разному. И естественно возникает вопрос, насколько хорошо работает приложение нашего сайта? Здесь следует пояснить, что значит «хорошо работает». В данном случае, я рассматриваю:

- скорость работы приложения;
- нагрузку на СУБД (частоту и сложность запросов);
- корректность возвращаемых пользователю данных.

Чтобы отслеживать эти и некоторые другие характеристики разрабатываемой программы, в Django имеется довольно удобный инструмент под названием:

Django Debug Toolbar

Давайте посмотрим, как его можно использовать для анализа работы нашего приложения.

Если перейти в репозиторий [pypi.org](#) и в поиске ввести «django debug toolbar», то увидим различные версии этого пакета. Установим последнюю из них в виртуальное окружение. Для этого, в терминале выполним команду:

```
pip install django-debug-toolbar
```

После установки, его нужно «прикрутить» к нашему сайту. Подробная информация о его настройке доступна на странице:

<https://django-debug-toolbar.readthedocs.io/en/latest/installation.html>

Согласно этой документации, в файле settings.py нужно зарегистрировать это приложение, добавив в INSTALLED_APPS строчку:

```
INSTALLED_APPS = [
    ...
    'debug_toolbar',
]
```

В этом же файле в коллекции MIDDLEWARE нужно прописать:

```
MIDDLEWARE = [
    ...
    'debug_toolbar.middleware.DebugToolbarMiddleware',
]
```

Далее, в этом же файле нужно прописать новую коллекцию INTERNAL_IPS со списком отслеживаемых IP-адресов

```
INTERNAL_IPS = [
    '127.0.0.1',
]
```

Переходим в файл urls.py и в режиме DEBUG нужно добавить маршруты для этого инструмента:

```
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
    ] + urlpatterns

    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Все, инструмент добавлен и давайте посмотрим, как он будет выглядеть и что показывать. Запускаем тестовый веб-сервер, переходим на главную страницу сайта и справа отображается панель Debug Toolbar.

Здесь мы видим версию Django, время формирования страницы, количество выполненных SQL-запросов, список используемых шаблонов и так далее. В целом, это довольно удобный и информативный инструмент. Если нужна более детальная информация, например, по SQL-запросам, то достаточно кликнуть по этому пункту и появится развернутая информация. В частности, здесь первые запросы связаны с авторизацией пользователя. Django их сформировал самостоятельно для обеспечения работы модуля авторизации. А ниже идут наши запросы, заданные на уровне ORM-команд. Разумеется, здесь мы их видим уже на уровне SQL-запросов к конкретной СУБД (SQLite).

Уникальные запросы нас вполне устраивают, они необходимы и не происходит их дублирования. А вот ниже видим однотипные запросы, что нехорошо. За что они отвечают. Смотрим запрос и видим, что идет чтение названия рубрики. Очевидно, это связано с обращением к категории в шаблоне index.html через переменную p:

```
<p class="first">Категория: {{p.cat}}</p>
```

Давайте проверим, так ли это. Уберем временно эту строчку, обновим страницу, и действительно, число запросов заметно уменьшилось. Почему это происходит? Вспоминаем, что в Django есть понятие «ленивых» и «жадных» запросов. Ленивые, то есть, отложенные запросы, которые выполняются только в момент непосредственного обращения к данным, например, для вывода рубрики. Как раз здесь выполняется отложенный запрос. Они достаточно удобны при работе с единичными записями, но когда их много, то возникает неприятный эффект резкого увеличения числа SQL-запросов, что мы сейчас и наблюдаем.

Как можно оптимизировать этот процесс? Очевидно, нужно сделать загрузку категорий и постов одним запросом. Для этого в Django имеются два полезных метода:

- select_related(key) – «жадная» загрузка связанных данных по внешнему ключу key, который имеет тип ForeignKey;
- prefetch_related(key) – «жадная» загрузка связанных данных по внешнему ключу key, который имеет тип ManyToManyField.

Как и где их использовать. Смотрите, за главную страницу у нас отвечает класс представления WomenHome и в нем определен метод get_queryset для выборки записей:

```
class WomenHome(DataMixin, ListView):
    ...
    def get_queryset(self):
        return Women.objects.filter(is_published=True)
```

Как раз здесь, после метода filter мы и запишем метод select_related:

```
def get_queryset(self):
    return Women.objects.filter(is_published=True).select_related('cat')
```


В этом методе указываем атрибут cat, который определен во вторичной модели Women как ForeignKey для связи с первичной моделью Category. Теперь, мы имеем «жадную» загрузку дополнительных данных из таблицы category и для получения названия рубрики нет необходимости делать запрос к БД.

Давайте посмотрим, как это будет работать. Возвращаемся в браузер, обновляем главную страницу и видим, что запросов действительно стало меньше. Вот так, благодаря Debug Toolbar и использованию «жадного» запроса мы оптимизировали работу нашего сайта.


Аналогично правим класс представления WomenCategory для оптимизации при отображении списка статей по отдельным рубрикам.


Как видите, Debug Toolbar довольно полезный инструмент для анализа работы сайта и я советую вам также использовать его или какой-либо аналогичный, чтобы создавать оптимизированные сайты.

Видео по теме





#1. Django - что это такое, порядок установки







#2. Модель MTV. Маршрутизация. Функции представления







#3. Маршрутизация, обработка исключений запросов,







#4. Определение моделей. Миграции: создание и выполнение







#5. CRUD - основы ORM по работе с моделями






#6. Шаблоны (templates). Начало





#7. Подключение статических файлов. Фильтры



←

→

←

→

←

→

←

→

←

→

←

→

←

→