

# Создание Web-сервисов на Python

Московский физико-технический институт и Mail.Ru Group

Неделя 4

# Содержание

<b>1</b>	<b>Python и WEB-фреймворки</b>	<b>2</b>
1.1	Архитектура web-фреймворка . . . . .	2
1.2	Примеры существующих фреймворков . . . . .	3
1.2.1	Django . . . . .	3
1.2.2	Flask . . . . .	3
1.2.3	Асинхронные веб-фреймворки . . . . .	3
<b>2</b>	<b>Django</b>	<b>4</b>
2.1	Роутинг и устройство view . . . . .	4
2.2	Установка и запуск простейшего приложения . . . . .	7
2.3	Шаблонизация в Django . . . . .	10
<b>3</b>	<b>Django ORM</b>	<b>16</b>
3.1	Работа с ORM . . . . .	16
3.1.1	Цепочка фильтров . . . . .	23
<b>4</b>	<b>HTML</b>	<b>30</b>
4.1	Панель разработчика в Chrome . . . . .	30
4.2	Основы HTML . . . . .	31

# 1. Python и WEB-фреймворки

## 1.1. Архитектура web-фреймворка

Системная архитектура — это организация и структура распределения элементов информационной системы, то есть это то, как логически распределена на наша система и как эти элементы взаимодействуют между собой. Если не следовать какой-то архитектуре, то в дальнейшем будет очень сложно вносить какие-то изменения и дорабатывать этот код. Поэтому люди начали придумывать некие паттерны разработки.

Собственно, один из них мы и рассмотрим — MVC (Model-View-Controller). Это схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента. Это сделано для того, чтобы если поменялся какой-то из компонентов, мы могли его поменять, и это изменение не затронуло Model-View.

Давайте рассмотрим, как происходит процесс обращения к клиенту.



Рис. 1

Клиент делает запрос к нашему приложению. Он попадает в контроллер. Контроллер как-то модифицирует данные с помощью модели. После этого модель сохраняет эти данные в базу. После этого модель предоставляет эти данные view. View как-то формирует ответ и отдает этот ответ клиенту.

Собственно, Controller. Контроллер отвечает за бизнес-логику, контроллер использует модели, для того чтобы как-то изменять, или добавлять, или удалять данные и представления, или view, чтобы их отображать. Собственно, в контроллере сосредоточена вся бизнес-логика нашей системы.

View, или представление, формирует ответ в конкретном формате, то есть во View попадают некие данные, и он их как-то отображает.

Модель отвечает за данные и правила их обработки, то есть все, что ты делаем с данными, мы делаем в моделях.

Рассмотрим на примере. Допустим, у нас есть модель «Студент» и нем поля «Имя» и «Оценка»,

и вьюха — это «Профиль студента». Рассмотрим контроллеры. Допустим, у нас есть контроллер «Посмотреть профиль». Собственно, мы хотим зайти на какую-то страничку и увидеть его профиль. Для этого нужно получить данные с помощью нашей модели «Студент» и с помощью View отобразить эти данные. Или, например, у нас есть контроллер «Поставить оценку». Собственно, для этого нам нужно прочитать модель «Студент», изменить «Оценку» также с помощью нашей модели «Студент» и вернуть данные с помощью нашего View. Собственно, как мы видим, контроллер отвечает за бизнес-логику нашего приложения: для получения данных он использует модели, для отображения он использует View.

## 1.2. Примеры существующих фреймворков

Рассмотрим web фреймворки.

### 1.2.1. Django

Слоган Django: это web фреймворк для перфекционистов с дедлайнами. Django простой, на нём написаны такие приложения, как Instagram, lamoda, Washington Post. По нему очень много документации, очень большое комьюнити, и в целом на нём очень приятно разрабатывать.

### 1.2.2. Flask

Flask — это очень простой минималистичный web фреймворк почти без ничего. Модели в него не включены, и для реализации модели используются сторонние инструменты. Flask идеально подходит для разработки простых сайтов.

### 1.2.3. Асинхронные веб-фреймворки

Это такие фреймворки, которые построены на кооперативной многозадачности. Это означает, что одним обработчиком в один момент времени может выполняться несколько задач.

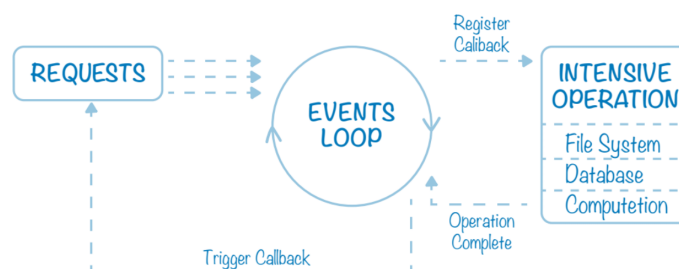


Рис. 2

Они основаны на event loop. Допустим, у нас есть несколько обработчиков, которые входят в сеть, например, обращаются по HTTP к какому-то сайту или ходят в базу. Есть некий процесс, который выполняют эти обработчики. Он натывается на хождение в сеть и мы эту операцию кладём в event loop и говорим, что когда нам данные придут из сети, верни управление обратно к нам в обработчик, а мы пока выполним другую операцию. Берём следующий какой-то кусок кода, натываемся опять на этот фрагмент, например, вхождения в сеть, и также кладём в event loop, и так далее. Потом в какой-то момент нам приходит ответ от сервера. Значит, ответ получили и возвращаем обратно управление в наш обработчик. Дальше выполняются какие-то операции и до тех пор, пока обработчик не закончится или не встретит опять такую же операцию, которые мы положим в event loop. Вот, собственно, на основе этого реализована кооперативная многозадачность.

Теперь рассмотрим примеры фреймворков. Первый — это Tornado. Это web фреймворк с реализацией асинхронной сетевой библиотеки. Асинхронность в целом позволяет держать намного больше нагрузки, чем синхронные web фреймворки.

Приложения на Tornado немного похоже на приложения на Flask, но есть нюансы. Например, у нас внутри нашего обработчика есть какая-то неблокирующая операция. Например, мы тут хотим подождать несколько секунд. И получается: к нам пришёл клиент, обратился к этому методу. Мы встретили операцию подождать несколько секунд, её положили в event loop, пришёл следующий клиент, мы обработали его запрос. Опять встретили, что нужно подождать несколько секунд, опять положили в event loop, и так далее. Прошло этих несколько секунд. event loop возвратил нам управление обратно в наш обработчик — и мы сформировали ответ клиенту. Тем самым мы смогли обработать намного больше запросов, чем бы это смог сделать синхронный web-фреймворк.

Другим примером асинхронного web фреймворка является Aiohttp. Точнее, это даже просто асинхронный HTTP-клиент-сервер на основе asyncio. Asyncio — это модуль, который обеспечивает инфраструктуру для написания программ с кооперативной многозадачностью.

## 2. Django

### 2.1. Роутинг и устройство view

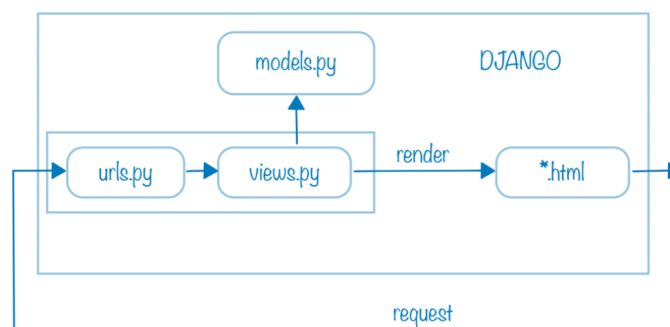


Рис. 3

Рассмотрим процесс обработки запроса: когда мы запрашиваем какой-то веб-сервис, то с самого начала запрос попадает в `urls`, там находит необходимый обработчик запроса, которому передается управление. Там выполняется бизнес-логика, формируется ответ клиенту и, собственно, передается клиенту.

Рассмотрим процесс, который происходит внутри `urls`. При старте приложения `django` загружает модуль, который указан в `root_urlconf`, который указан в `settings`. А потом внутри него ищет переменное `urlpatterns`. Когда `django` приходит запрос, она проверяет по порядку каждое регулярное выражение, которое указано в `urlpatterns`. В случае, если он нашел совпадение, то вызывает соответствующий этому `url`'у обработчик, который передает объект `httprequest`'а и параметр, который нашло регулярное выражение. Если он просмотрел все `url`'ы и не нашел совпадения, то `django` вызывает соответствующий обработчик ошибки.

Давайте рассмотрим процесс роутинга. Это сопоставление `url`'а с его обработчиком. Это описано в файликах `urls` и переменной `urlpatterns`. `Urlpatterns` представляет из себя массив из объектов `urls`. Когда мы создаем объект `urls`, то мы указываем там регулярное выражение, которое соответствует `url`'у и его обработчик.

Давайте рассмотрим на примере.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$',
        views.month_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$', views.article_detail),
]

# /articles/2003/ - views.special_case_2003(request)
# /articles/2004/ - views.year_archive(request, '2004')
# /articles/2005/12/ - views.month_archive(request, '2005', '12')
# /articles/2003/03/03/ - views.article_detail(request, year='2003', month='03', day='03')
```

Рис. 4

Например, мы делаем запрос какому-то сервису `/articles/2003/`. Мы тестируем первый `url`, видим, что он совпадает, будет вызван обработчик с параметром `request`. Теперь давайте рассмотрим запрос `/articles/2004/`. Мы тестируем первое регулярное выражение, видно, что оно не подходит, тестируем второе — видно, что оно подходит. Управление передастся второму обработчику с `request` и с одним найденным параметром `2004`. То же самое произойдет и в третьем примере. Следует рассмотреть четвертый пример, где в группах указано их имя. И когда мы, например, сделаем запрос `/articles/2005/12/01`, то вызовется соответствующий обработчик с параметрами `request` и позиционными параметрами `year`, `month` и `day`.

Важно отметить, что, если внутри регулярного выражения были именованные аргументы, то неименованные будут игнорироваться. Найденные аргументы всегда будут строки — это надо учитывать при написании `view`. Каждое регулярное выражение компилируется только один раз, и это не влияет на производительность.

Рассмотрим `include`. С помощью `include` мы можем организовать иерархическую структуру: мы указываем в переменной `urlpatterns` такой `url`, который в начале соответствует префиксу, и далее мы указываем `include`, либо переменную, которая содержит список из `url`'ов, либо файл, в котором есть переменная `urlpatterns`. Допустим, у нас есть множество `url`'ов с одинаковым префиксом, и очень длинным, и в котором указываются некоторые параметры. И этот префикс содержится во множестве `url`'ов, которые не хочется повторять. Для этого мы делаем `url`, в котором мы указываем этот префикс и дальше пишем `include`, а внутри этому `include`'у передаем список `url`'ов, в котором указаны только те части, которые отличаются. После того, как мы с помощью `urls` определили обработчик, которому следует передать обработку этого запроса, он будет вызван.

Обработчик, то есть `view`. Каждое `view` принимает объект `HttpRequest` в качестве своего первого параметра, который обычно называется в коде `request`. Также каждое `view` должна возвращать объект `HttpResponse`, который содержит сгенерированный ответ.

Давайте поподробнее рассмотрим объект `HttpRequest`. Нам интересны его атрибуты, такие как `method`, в котором содержится метод `Http` запроса; атрибут `GET`, в котором содержатся `get` параметры; атрибут `POST`, в котором содержится тело запроса; и атрибут `FILES`, в котором содержатся файлы, которые переданы с запроса. Также атрибут `COOKIES`, в котором указаны `cookie` запросы, и атрибут `META`, в котором указаны заголовки нашего запроса.

Теперь `HttpResponse`. Рассмотрим его конструктор, собственно, когда мы формируем тело ответа, мы создаем `HttpResponse` и его возвращаем:

- `content`, это тело нашего ответа,
- `content_type` — это тип `MIME`, тип данных, которые мы будем возвращать,
- `status` — это статус нашего `http` ответа, например, `ok`, `not found`, `bad request` и так далее,
- `reason` — это в случае, если мы хотим как-то предопределить `reason` в `HTTP`,
- `charset` — это собственная кодировка, с которой будут переданы данные клиенту.

Для того, чтобы установить некоторые заголовки в `HttpResponse`, можно с ним работать так же, как со словарем. То есть мы указываем ключик и присваиваем ему какое-то значение. Чтобы удалить этот заголовок, поступаем так же, как в словаре. Если мы хотим проверить, существует ли наш заголовок в этом ответе, то мы вызываем метод `has header`. Если нам нужно установить или удалить `cookie`, нам нужно вызвать методы `set cookie` и `delete cookie` соответственно. В случае, если нам нужно в процессе обработки запроса формировать ответ, то можно вызвать у `HttpResponse`'а метод `write` и дописывать ответ.

```

response = HttpResponse()
# установка заголовка
response['Age'] = 120
# удаление заголовка
del response['Age']
# case-insensitive проверка наличия заголовка
response.has_header('Age')
# установка cookie
response.set_cookie(key, value, max_age, expires, path,
                    domain, secure, httponly)
# удаление cookie
response.delete_cookie(key, path, domain)
# дописать
response.write("<p>Here's the text of the Web
page.</p>")
response.write("<p>Here's another paragraph.</p>")

```

Рис. 5

Теперь рассмотрим декораторы и их применение в веб-приложении. Рассмотрим пример. Допустим, в множестве запросов нам нужно всегда проверять, что Http method, с которым пришел запрос, является get. То каждый раз нам придется во view писать условие if request method не равняется get return http method not allowed. В этом случае удобно применить декоратор. Мы просто делаем декоратор, в котором добавляем логику проверки, что нам пришел get запрос, и вешаем этот декоратор на все наши обработчики. В django'е поставляется несколько стандартных декораторов, например, тот, что мы уже рассмотрели, это require http method, которому можно передать список методов, которым разрешена обработка этой view. Или например short cut'ы. require\_GET, который разрешает только обработку get запросов, или require\_POST, который разрешает обработку только post запросов.

```

def some_decorator(some_decorator_params):
    def decorator(func):
        @functools.wraps(func)
        def inner(request, *args, **kwargs):
            # do something
            return func(request, *args, **kwargs)
        return inner
    return decorator

@some_decorator('some_params')
def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)

```

Рис. 6

## 2.2. Установка и запуск простейшего приложения

Для начала давайте создадим проект. Перейдем в нужную нам папку, где мы хотим создавать наше приложение, и выполним команду django-admin startproject и название нашего проекта.



Давайте его назовем `coursera_blog`. Мы видим, что Django создала нам наш проект, перейдем в него. Я крайне рекомендую устанавливать виртуальное окружение, так как у вас может быть множество проектов с разными версиями Django и с разными зависимостями.

Для этого напишем `virtualenv`, укажем версию Питона, `python3`, и назовем ее `env`.

Перейдем внутрь этого окружения. Также я крайне рекомендую создавать файл зависимости `requirements.txt`. Укажем там нашу первую зависимость — это Django версии 1.11. Для того чтобы поставить зависимости из файла, необходимо написать `pip install -r requirements`. Собственно, теперь у нас Django установлен в наше виртуальное окружение.

Давайте попробуем запустить наш проект. У нас запустился наш dev сервер на IP адресе 127.0.0.1 на порту 800.

```
MacBook-Pro-Alexander:coursera_blog alexander$ source ./env/bin/activate
(env) MacBook-Pro-Alexander:coursera_blog alexander$
(env) MacBook-Pro-Alexander:coursera_blog alexander$ vim requirements.txt
(env) MacBook-Pro-Alexander:coursera_blog alexander$
(env) MacBook-Pro-Alexander:coursera_blog alexander$ pip install -r requirements.t
xt
Collecting django==1.11 (from -r requirements.txt (line 1))
  Using cached Django-1.11-py2.py3-none-any.whl
Collecting pytz (from django==1.11->-r requirements.txt (line 1))
  Using cached pytz-2017.3-py2.py3-none-any.whl
Installing collected packages: pytz, django
Successfully installed django-1.11 pytz-2017.3
(env) MacBook-Pro-Alexander:coursera_blog alexander$
(env) MacBook-Pro-Alexander:coursera_blog alexander$
(env) MacBook-Pro-Alexander:coursera_blog alexander$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until you a
pply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

December 03, 2017 - 09:31:01
Django version 1.11, using settings 'coursera_blog.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Рис. 7

Давайте теперь создадим наше первое приложение. Установим наш dev server, и для того чтобы создать приложение в Django, необходимо написать `python manage.py startapp core`. Создалась папка `core`.

Рассмотрим, что же внутри нее. Views — это контроллеры. В Django немного переименовали парадигму `model view controller`, `model` остался без изменений, `view` переименовали в `template`, а контроллер переименовали во `view`. То есть папка Views — это папка с контроллерами. Папка `models` — это папка с нашими моделями. Папка `migrations` — это папка с миграциями. Когда мы добавили новое приложение, его необходимо указать в `installed apps`.

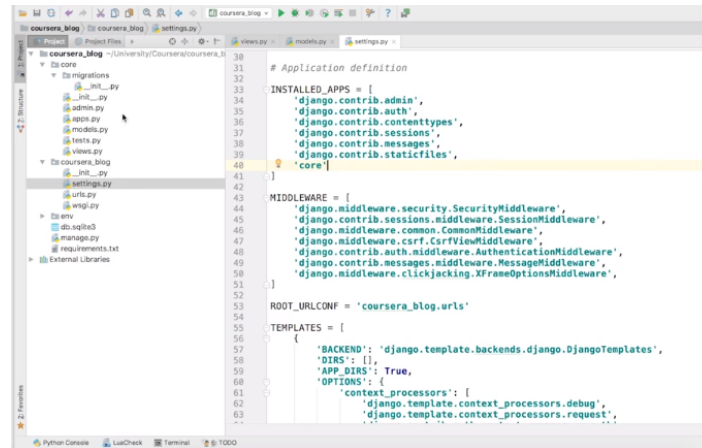


Рис. 8

И давайте напишем нашу первую простейшую View. Для этого в файлике views создадим функцию index, которая принимает один параметр request и возвращает http response.

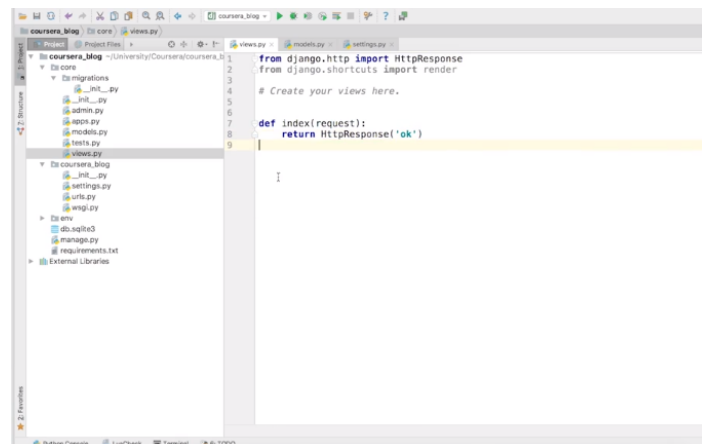


Рис. 9

И напишем наш первый роут index. Для этого в корневой файлик urls добавим наш view. Это будет означать, что когда мы запустим наш dev server, то по /index нам откроется view.

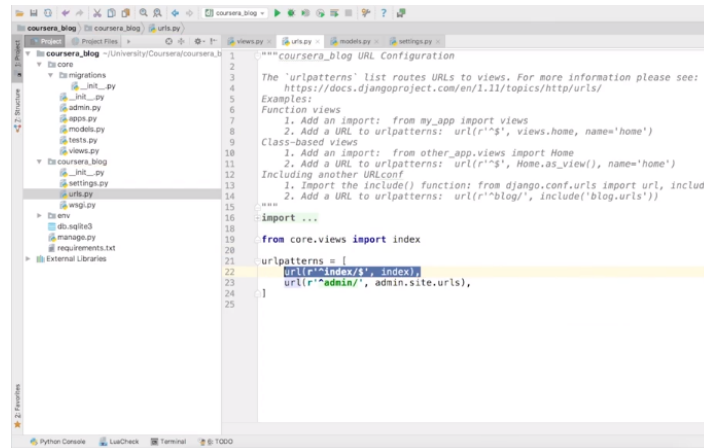


Рис. 10

Запускаем наш dev server и открываем его `http://127.0.0.1:8000/index`. Нам отдастся ок.

## 2.3. Шаблонизация в Django

Шаблонизация — необходимый способ генерации HTML динамически для формирования HTTP ответа. Наиболее распространенный подход основан на шаблонах.

У нас есть некий сайт, мы зашли на его главную страницу, и там в шапке, например, есть имя и фамилия, когда вы зарегистрировались. Эта страничка не статическая, то есть мы не можем просто взять файл, написать туда нашу HTML верстку и ее отдавать. Нам нужно динамически для каждого пользователя формировать этот HTML, и для этого и нужен шаблонизатор. Django может быть настроен с использованием одного или нескольких движков шаблонизации. В Django встроен свой механизм шаблонизации, это Django template language. Также можно использовать какие-то альтернативные движки, например, Jinja 2.

Настройка: в файле `settings.py` есть переменная `TEMPLATES`, в нем указываются все параметры для движков шаблонизации, рассмотрим эти параметры. `BACKEND` — это путь к движку шаблонизации, `DIRS` — это папки, в которых будут искаться все наши шаблоны, `APP_DIRS` — это параметр, который, если он установлен в `true`, говорит, что следует искать шаблоны внутри наших приложений.

Рассмотрим пример. Создадим папку `templates` на уровне приложений и внутри каждого приложения также создадим `templates`. Будем размещать все наши шаблоны, которые относятся к этому приложению внутри этого приложения, а все базовые шаблоны, которые используются везде, по всему проекту, мы будем размещать в корневой папке `templates`. Параметр `Options`: в нем указываются параметры, специфичные для каждого из движков.

```
polls/
  __init__.py
  models.py
  templates/
    poll.html
  views.py
templates/
  base.html
```

Рис. 11

Давайте теперь рассмотрим пример вьюхи, которая выполняет бизнес-логику и в итоге берет шаблон, рендерит его и отдает нам контекст. После выполнения бизнес-логики нам нужно получить шаблон по его пути. Это делается с помощью `loader.get_template`. Мы получили этот шаблон, потом нам нужно указать контекст, на основе которого мы будем рендерить этот шаблон. После этого мы делаем `template.render`, передаем туда наш контекст и параметры `request`, после того как шаблон отрендерится, в результате мы получим просто некий текст и передадим его в контент `HttpResponse`.

```
from django.http import HttpResponse
from django.template import loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    context = {'foo': 'bar'}
    return HttpResponse(t.render(context, request))
```

Рис. 12

Каждый раз это делать неудобно, поэтому в Django придумали шорткаты. Шорткат `render`, который позволяет указать `request` имя шаблона и его контекст, чтобы отрендерить и отдать клиенту этот ответ, и `redirect`, например, если мы хотим отправить ленту на какой-то другой URL.

```
render(request, template_name, context,
content_type, status, using)

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html',
{'foo': 'bar'},)

redirect(to, permanent, *args, **kwargs)

def my_view(request):
    ...
    return redirect('/some/url/')
```

Рис. 13

Рассмотрим синтаксис шаблонизатора.

Начнем с передачи переменных вовнутрь шаблона. Есть некий шаблон, который справедлив для всех пользователей. При рендере этого шаблона останется `My first name is` и поставится имя пользователя. Так же там `My last name is`. Дальше ниже мы видим обращение к атрибутам, то есть так же, как и в Питоне, можно обращаться к атрибутам каких-то сложных объектов. Например, если это `my_dict`, некий словарь, то мы можем обращаться к его ключам через точку, и к атрибутам объекта тоже можем обращаться через точку. К индексам в списке мы можем обращаться тоже через точку и указываем необходимые нам индексы.

```
My first name is {{ first_name }}.
My last name is {{ last_name }}.

{{ my_dict.key }}
{{ my_object.attribute }}
{{ my_list.0 }}
```

Рис. 14

Условия, например, нужны тогда, когда нам нужно отобразить различный контент для различных пользователей. Для этого существует тег `if` и, например, тег `for` — он служит для вывода некоторого массива. Например, у нас есть список топиков и нам нужно вывести этот список в шаблоне. Мы пишем цикл `for`, указываем в теле этого цикла один наш топик, и он будет отрендерен.

```
{% if number == 2 %}
    Число равно двум.
{% elif number == 3 %}
    Число равно трем.
{% else %}
    Значение числа: {{ number }}
{% endif %}

{% for o in some_list %}
    {{ o }}
{% endfor %}
```

Рис. 15

Фильтры и теги. В Django шаблонизаторе есть достаточное количество стандартных фильтров и тегов, которые можно использовать, но иногда требуется какой-то специфический функционал, который нужно реализовать самому и использовать его внутри шаблонов. Для этого необходимо создать внутри вашего приложения папку `templatetags` и там указать модуль, например, `poll_extras`. Чтобы этот модуль использовать в дальнейшем внутри шаблона, необходимо с помощью `load` загрузить этот модуль.

Рассмотрим пример использования фильтра. Например, нам нужно сначала преобразовать строку к нижнему регистру. Мы объявляем свой фильтр `lower`, его применяем к нашей строке с помощью вертикальной черточки, после нее пишем имя нашего фильтра. Этот фильтр, который преобразует строку к нижнему регистру, пишем в файлике `poll_extras` и вешаем на него декоратор `register.filter`.

```
from django import template

register = template.Library()

@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()

{{ somevariable|cut:"0" }}
{{ somevariable|lower }}
```

Рис. 16

Теги имеют немного другое предназначение и другой синтаксис. В то время как фильтры применяются к некоторым объектам, теги могут использоваться вообще без параметров или с некоторыми параметрами. Для того чтобы зарегистрировать такой тег, нужно его также объявить в файле `poll_extras` и повесить на него декоратор `register.simpletag`.

```

@register.inclusion_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...

{% my_tag 123 "abcd" book.title
warning=message|lower profile=user.profile %}

```

Рис. 17

Inclusion теги – в целом то же самое, что и обычные теги, только обычный тег возвращает некоторое значение, например, был у нас тег `sum`, который суммирует два числа, и он возвращает сумму этих чисел. А `inclusion` тег применяется тогда, когда нужно вернуть верстку. Например, в Django есть стандартный тег `CSRF-Token`, но при применении такого токена он возвращает верстку поля `Input`.

Теперь рассмотрим, как же следует работать с URL'ами в шаблонах. Для этого для начала следует рассмотреть, что такое `namespace` и `name` URL. Давайте укажем в `include` его `namespace`, а в `URL` его `name`. И в дальнейшем можно получать URL на основе его `namespace` и на основе его имени, как это представлено на примере.

```

# urls.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^author-polls/',
        include('polls.urls', namespace='author-
polls')),
]

# polls/urls.py
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(),
        name='index'),
    url(r'^(?P<pk>\d+)/$',
        views.DetailView.as_view(), name='detail'),
]

```

Рис. 18

Допустим, вы не стали использовать механизм резолвинга URL и везде по всему проекту написали просто строками ваши URL. В случае с резолвингом вы придумали некие `namespace` и некие имена, которые уже никогда не поменяются. Вы просто идете в файл `urls.py`, меняете на нужный ваш URL, и он автоматически меняется по всему проекту.

Теперь давайте рассмотрим наследование шаблонов. Допустим, у вас есть некий контентный сайт, где есть `header` и есть `footer` одинаковые, только отличается середина сайта. Самый простой вариант, который может прийти в голову, это просто во всех шаблонах будем добавлять

эти общие части. Но это неудобно. Поэтому придумали механизм наследования. Мы делаем некий базовый шаблон, где описываем все базовые компоненты нашего сайта и указываем блоки, которые можно переопределить с помощью тега `block`. А дальше в шаблонах наследниках мы указываем `extend`, с какого шаблона нам наследоваться, и переопределяем также с помощью тега `block` те места, которые мы хотим изменить.

```
{# blog.html #}

{% extends "base.html" %}

{% block title %}My amazing blog
{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

Рис. 19

И `include` — это в целом почти такой же механизм, как `extend`, но немного другой. Например, у нас есть некий компонент, который используется повсеместно на сайте, например, верстка топика. И нам нужно этот топик вставить в нескольких местах, например, в истории написания топиков пользователя и, например, в его ленте. Мы делаем некий шаблон топика и потом с помощью тега `include` инклюдим этот топик и передаем ему какие-то параметры.

```
{# blog.html #}

{% extends "base.html" %}

{% block title %}My amazing blog{%
endblock %}

{% block content %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
        {% include "entry.html" with
entry=entry %}
    {% endfor %}
{% endblock %}
```

Рис. 20



## 3. Django ORM

### 3.1. Работа с ORM

До этого момента мы не рассматривали хранение данных пользователя и в дальнейшем выдачу этих данных обратно пользователю. Это делается с помощью базы данных.

Есть множество способов, как это можно реализовать в коде.

Один из них — это писать сырой SQL-код. Это крайне неудобно:

- монотонное, однотипное написание SQL-кода;
- SQL — это стандарт, но у каждой БД он чуточку свой, и в случае, если мы захотим как-то поменять базу или что-то в этом роде, у нас это просто так не получится;
- генерация миграций. Например, вы добавили какое-то поле, и вам это поле нужно добавить в базу. И например, вы не один разрабатываете проект, а вас много, и после того, как вы добавили это поле, вам нужно эту SQL миграцию отправить всем своим коллегам, они ее тоже применят, и потом, когда ваш код будет раскладываться на боевой сервер, вам также эту миграцию нужно применить руками;
- уа уровне кода сложно понять структуру данных.

Решение. Все таблицы, которые у нас есть в БД, описываются с помощью Python классов. Выполнение запросов к БД будем производить через вызов методов данного класса. Каждая запись в таблице — это объект данного класса. И на основе изменений наших Python классов мы будем генерировать автоматические миграции.

Рассмотрим процесс настройки. Для этого в файлике settings.py есть переменная DATABASES.

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'mysql.connector.django',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'OPTIONS': {
            'autocommit': True,
        }
    }
}
```

Рис. 21


Там указываются все базы данных, к которым подключено наше приложение. Для того, чтобы указать дефолтную базу данных, мы в нашем дикте указываем ключик default и указываем

параметры. Собственно, ENGINE — это тот коннектор, который будет использоваться для подключения к базе. NAME — это имя базы, к которой мы будем подключаться. USER — это имя пользователя, с которым мы будем подключаться к базе. PASSWORD — это пароль, с которым мы будем подключаться к этой базе. И, собственно, HOST — это где расположена наша база. Также есть параметр PORT — на какой порт нужно стучаться. В этом случае он просто возьмет дефолтный. И также параметр OPTIONS, он специфичен для каждой базы данных.

Рассмотрим на примере. Допустим, у нас есть некоторая модель Person.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```



The diagram illustrates the mapping from Django code to a database table and a file structure. On the left, a SQL statement is shown: `CREATE TABLE myapp_person ( "id" serial NOT NULL PRIMARY KEY, "first_name" varchar(30) NOT NULL, "last_name" varchar(30) NOT NULL );`. On the right, a file structure is shown for `myapp/`, including `__init__.py`, `admin.py`, `apps.py`, `migrations/`, `__init__.py`, `models.py` (highlighted), `tests.py`, `urls.py`, and `views.py`.

Рис. 22

Мы описываем поле `first_name` и `last_name`. Это нужно делать в файлике `models.py`. Собственно, по этой нашей модели будет сгенерирован запрос в базу данных `CREATE TABLE`, и там будет указан `id`, `first_name` и `last_name`. И это нам не приходится делать руками, это все за нас делает автоматически Django.

Теперь давайте рассмотрим поля модели. Существует множество типов полей: Boolean, Char и т.д. Django использует их для определения типа столбца. То есть мы знаем, что у столбца есть тип в базе данных, например, INTEGER, VARCHAR или TEXT. Собственно, на основе этого класса она и определяет тип поля. Виджет — это HTML виджет, который будет использоваться при отображении. И минимальные требования к проверке данных, которые переданы в это поле.

Рассмотрим отдельно типы полей, которые помогают организовывать связь между моделями. Сначала рассмотрим Many2one связь. Для этого существует тип поля `ForeignKey`.

Рассмотрим пример. Допустим, у нас есть модель «производитель» и модель «машина». У одного производителя может быть множество машин. Собственно, мы указываем поле «производитель» в модели машины и пишем там `models.ForeignKey`, указываем ссылку на производителя и говорим, что делать в случае удаления производителя. То есть каскадно удаляем все машины.

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer =
models.ForeignKey(Manufacturer,
on_delete=models.CASCADE)
    # ...
```

Рис. 23

Рассмотрим теперь Many2many связь. Для ее организации также есть свой тип поля, это ManyToManyField. Собственно, в нем также указываем ссылку на ту модель, с которой мы хотим организовать Many2many связь.

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

Рис. 24

Теперь рассмотрим параметры полей:

- `null` — если он установлен в `True`, то Django будет хранить пустые значения;
- `blank`. Если он установлен в `True`, то поле может быть пустым. Тонкая грань существует между `null` и `blank`. `null` — это именно `null` в базе, а `blank` — это то, что Django разрешит принять пустые данные;
- `db_index`. В случае, если он установлен в `True`, то будет построен индекс по этому полю;
- `default` — это, собственно, дефолтное значение этого поля;
- `primary_key`, если он установлен в `True`, то это поле будет в дальнейшем считаться первичным ключом;
- `unique` — если он установлен в `True`, то будет построен уникальный индекс, и это поле будет уникальным по всей таблице;

- `validators` — это список валидаторов для этого поля.

Миграция — это способ применения изменений, которые делаются в моделях, в схему базы данных. Это делается с помощью следующих команд:

- `makemigrations` создает файл миграции. То есть вы добавили какое-то поле, он определил, какое, и сгенерировал такой файл, в котором описаны все действия, которые нужно сделать с базой;
- `migrate` — он берет этот файл и применяет его к базе;
- `sqlmigrate` — он отобразит тот SQL, который будет применен во время миграции;
- `showmigrations` — он отобразит список всех миграций и их статус.

Рассмотрим workflow. Собственно, вы добавили какое-то поле, сделали `python manage.py makemigrations`, он вам сгенерировал файл. После этого вы написали `python manage.py migrate`, он взял этот файл и применил его к вашей базе данных. Например, вы добавили какое-то поле, выполнили этот процесс, отправили ваш код в GitHub, ваш коллега получил этот проект и также выполнил `migrate`, и у вас базы консистентны: и у вас, и у вашего коллеги.

Теперь рассмотрим простые запросы: создание, редактирование и удаление. Для этого объявим три модельки. Это `Blog`, `Author` и `Entry`.

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

class Entry(models.Model):
    blog = models.ForeignKey(Blog,
on_delete=models.CASCADE)
    headline =
models.CharField(max_length=255)
    authors = models.ManyToManyField(Author)
```

Рис. 25

Собственно, для того, чтобы добавить `Blog`, нужно создать объект класса `Blog` и передать в него необходимые кварги, и после этого сделать `save`. Он сгенерирует SQL запросы и выполнит их в базе.

```
from blog.models import Blog

# создание
b = Blog(name='Beatles Blog', tagline='All
the latest Beatles news.')
b.save()

# изменение
b.name = 'New name'
b.save()
```

Рис. 26

Если же мы хотим изменить какое-то поле, мы присваиваем полю какое-то новое значение и делаем save. Он определит, что такая запись существует уже в базе и сгенерирует не insert, а update. В случае, если мы хотим удалить, мы просто у объекта делаем delete, и наша модель удаляется, и, собственно, строка из базы тоже удаляется.

Теперь рассмотрим сохранение ForeignKey и ManyToManyField. Например, мы получаем некоторое Entry, мы хотим поменять в нем Blog. Собственно, мы получаем Entry, берем entry.blog, меняем на нужное значение и делаем save. Собственно, мы поменяли связь. Для ManyToMany связи все даже проще. Если мы хотим Entry соединить с какими-то авторами, мы получаем объекты этих авторов и делаем entry.authors.add и добавляем этих авторов.

```
from blog.models import Blog, Entry, Author

# fk
entry = Entry.objects.get(pk=1)
cheese_blog = Blog.objects.get(name="Cheddar
Talk")
entry.blog = cheese_blog
entry.save()

# m2m
joe = Author.objects.create(name="Joe")
paul = Author.objects.create(name="Paul")
entry.authors.add(joe, paul)
```

Рис. 27

Для извлечения данных из базы данных нужно создать QuerySet. QuerySet представляет собой набор объектов из базы.

QuerySet можно получить с помощью менеджеров модели. Каждая модель имеет, по крайней мере, один менеджер, и по умолчанию он называется objects. Собственно, давайте сначала посмотрим, как же нам получить все объекты из базы. Для лучшего понимания будем сопоставлять

наши запросы в ORM с SQL-запросами.

Для того чтобы получить все объекты, нужно обратиться к классу нашей сущности `.objects`, вызвать то есть его менеджер и после этого сделать `.all`. В итоге мы получим `QuerySet` и обратимся к полю `Query` и посмотрим, какой же SQL-запрос он делает под капотом. Он соответствует простому SQL-выражению `select` перечисления наших полей `from` таблица. Стоит заметить, что класс называется `entry`, а таблица называется `core entry`. `Core` — это название приложения. По дефолту Django называет именно так модели, то есть в начале имя приложения, потом имя сущности.

```
>>> all_entries = Entry.objects.all()
>>> print(all_entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry"
```

Рис. 28

Для того чтобы получить только одну сущность, есть метод `get` у менеджера. Для этого нужно потом еще передать условия с помощью кваргов, например, `get pk = 1`. Это соответствует выражению `select` перечисление полей `from` таблица `where` условия `core entry.id = 1`. Если же это SQL-выражение вернет больше сущностей, чем одна, то будет исключение. Если ничего не вернет, то тоже будет исключение.

```
>>> Entry.objects.get(pk=1)
SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry"
WHERE "core_entry"."id" = 1
```

Рис. 29

Допустим, мы хотим получить все `entry` с хедлайном `name`. Для этого необходимо написать `entry.objects.filter` и передать кварги `headline = name`. В итоге это соответствует запросу `select` перечисление полей `from` наша таблица `where headline = name`.

```
>>> entries =
Entry.objects.filter(headline='name')
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry" WHERE
"core_entry"."headline" = name
```

Рис. 30

Также есть метод у менеджера exclude, то есть не включать объекты с таким условием. То есть, например, мы хотим получить все entry, у которых headline не равняется name.

```
>>> entries =
Entry.objects.exclude(headline='name')
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry" WHERE NOT
("core_entry"."headline" = name)
```

Рис. 31

Что же делать, если нам необходимо передать несколько условий? Самое простое — это передать несколько кваргов в фильтр. Это соответствует условию end в SQL.

```
>>> entries =
Entry.objects.filter(headline='name',
blog_id=1)
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline" FROM
"core_entry" WHERE
("core_entry"."headline" = name AND
"core_entry"."blog_id" = 1)
```

Рис. 32

Что же делать, если нам необходимо условие or? Для этого существует класс Q. Например, мы хотим, чтобы headline равнялся либо name, либо name2, но при этом blog\_id = 1.

```
>>> from django.db.models import Q
>>> entries = Entry.objects.filter(
...     (Q(headline='name') |
...      Q(headline='name2'))
...     & Q(blog_id=1)
... )
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry"
WHERE (
("core_entry"."headline" = name OR
"core_entry"."headline" = name2)
AND "core_entry"."blog_id" = 1)
```

Рис. 33

### 3.1.1. Цепочка фильтров

Так как метод фильтра возвращает QuerySet и к QuerySet также применить фильтр, то можно составлять цепочки фильтров. Например, мы сначала хотим пофильтровать по `headline = name`, а потом еще исключить все entry, у которых `blog_id = 1`. Это можно сделать в два этапа.

```
>>> entries =
Entry.objects.filter(headline='name')
>>> entries =
entries.exclude(blog_id=1)
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry"
WHERE ("core_entry"."headline" = name
AND NOT ("core_entry"."blog_id" = 1))
```

Рис. 34

Мы рассмотрели сейчас условия только с равенством, то есть равно или не равно. Как же нам сделать какие-то сложные условия типа больше, меньше или in и так далее? Для этого существует Lookups, это условие, которое будет указано в SQL-выражении where. Lookups можно передавать методу filter, exclude и get в формате названия `field__lookuptype = value`. Это также



кварг-параметры. Например, мы хотим получить все entry, у которых `blog_id = 1` или `2`, для этого мы бы применили оператор `in` в SQL.

```
>>> entries =
Entry.objects.filter(blog_id__in=[1,
2])
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline" FROM
"core_entry" WHERE
"core_entry"."blog_id" IN (1, 2)
```

Рис. 35

Lookuptypes:

- `contains`, `icontains` — `WHERE headline LIKE, ILIKE '%<запрос>%'`;
- `startswith`, `istartswith` — `WHERE headline LIKE, ILIKE '<запрос>%'`;
- `endswith`, `iendswith` — `WHERE headline LIKE, ILIKE '%<запрос>'`;
- `in` — `WHERE id IN (...)`;
- `gt`, `gte`, `lt`, `lte` — `WHERE id >, >=, <, <= 4`;
- `range` — `WHERE blog_id BETWEEN 5 and 10`;
- `isnull` — `WHERE headline IS NULL`.

Рассмотрим теперь фильтрацию по связанной сущности. Например, мы хотим получить все entry, в которых название блога какое-то.

```
>>> entries =
Entry.objects.filter(blog__name=
'Beatles Blog')
>>> print(entries.query)

SELECT "core_entry"."id",
"core_entry"."blog_id",
"core_entry"."headline"
FROM "core_entry"
INNER JOIN "core_blog" ON
("core_entry"."blog_id" =
"core_blog"."id")
WHERE "core_blog"."name" = Beatles
Blog
```

Рис. 36

Рассмотрим такой случай, что нам, например, нужно пофилтровать блоги, у которых авторы entry имеют название «Леннон». Смотрите, насколько трудно это сделать в SQL — даже не трудно, а насколько долго.

```
>>> entries =
Blog.objects.filter(entry__authors__name='Lennon')
>>> print(entries.query)

SELECT "core_blog"."id", "core_blog"."name",
"core_blog"."tagline"
FROM "core_blog"
INNER JOIN "core_entry" ON ("core_blog"."id" =
"core_entry"."blog_id")
INNER JOIN "core_entry_authors" ON
("core_entry"."id" =
"core_entry_authors"."entry_id")
INNER JOIN "core_author" ON
("core_entry_authors"."author_id" =
"core_author"."id")
WHERE "core_author"."name" = Lennon
```

Рис. 37

Но есть нюансы с использованием reverse-relationship. Давайте их рассмотрим.

Первый — это нюанс дублирования. Например, у нас есть два блога, мы их посмотрим с помощью операции `blog.objects.all` — у нас вывелось два блога, `name1` и `name2`. И два entry. Напишем такой запрос, который выведет все блоки, у которых есть entry с headline, который содержит headline. В результате этого запроса будет выведено две одинаковые сущности: blog с `name1` и blog с `name1`. Когда прижойним таблицу blog к entry и поставим такую сущность, на самом деле SQL также нам вернет две сущности, два блога. И этот результат смайлся в объекты и нам выдался. Как это решить? Есть оператор `distinct`. Если мы после такого фильтра напишем `distinct`, нам выведется только одна сущность.

```
# setup
>>> Blog.objects.all()
<QuerySet [<Blog: name1>, <Blog: name2>]>
>>> Entry.objects.all()
<QuerySet [<Entry: blog: name1 headline1>, <Entry: blog:
name1 headline2>]>
# нюанс
>>>
Blog.objects.filter(entry__headline__contains='headline')
<QuerySet [<Blog: name1>, <Blog: name1>]>

# как решить
Blog.objects.filter(entry__headline__contains='headline')
.distinct()
<QuerySet [<Blog: name1>]>
```

Рис. 38

Второй нюанс связан с `isnull`. Допустим, у нас есть два блога: `name1` и `name2` и только один `entry`. У блога `name1` с `headline` `None`. И, допустим, нам необходимо написать запрос, который выведет все блоги, у которых `headline` `null`. В результате мы получим не один блог `name1`, как мы ожидали, а два. Django посчитало, что так как у блога `name2` не существует вообще связанной сущности `entry`, то это тоже `isnull`, и его включило в запрос. Необходимо добавить еще одно условие: `entry__isnull=False`, то есть существует связанная сущность. И тогда мы получим правильный результат.

```
# setup
>>> Blog.objects.all()
<QuerySet [<Blog: name1>, <Blog: name2>]>
>>> Entry.objects.all()
<QuerySet [<Entry: blog: name1 None>]>
# нюанс
>>> Blog.objects.filter(entry__headline__isnull=True)
<QuerySet [<Blog: name1>, <Blog: name2>]>

# как решить
>>> Blog.objects.filter(entry__isnull=False,
entry__headline__isnull=True)
<QuerySet [<Blog: name1>]>
```

Рис. 39

Для того чтобы посортировать нашу выборку, необходимо сделать `order by`. Для того чтобы получить только первые пять записей, нужно так же, как с массивами в «Питоне» применить оператор `slice`.

```
>>> blogs = Blog.objects.all().order_by('name')
>>> print(blogs.query)

SELECT "core_blog"."id", "core_blog"."name",
"core_blog"."tagline"
FROM "core_blog"
ORDER BY "core_blog"."name" ASC

>>> blogs = Blog.objects.all().order_by('-name',
'tagline')
>>> print(blogs.query)

SELECT "core_blog"."id", "core_blog"."name",
"core_blog"."tagline"
FROM "core_blog"
ORDER BY "core_blog"."name" DESC, "core_blog"."tagline"
ASC
```

Рис. 40

```
>>> entries = Entry.objects.all()[:5]
>>> print(entries.query)
SELECT "core_entry"."id",
"core_entry"."blog_id", "core_entry"."headline"
FROM "core_entry" LIMIT 5

>>> entries = Entry.objects.all()[5:10]
>>> print(entries.query)
SELECT "core_entry"."id",
"core_entry"."blog_id", "core_entry"."headline"
FROM "core_entry" LIMIT 5 OFFSET 5
```

Рис. 41

Вы можете сколько угодно конструировать QuerySet, Django выполнит запрос только тогда, когда будет вызвана операция обращения к данным (evaluation).

- Iteration. QuerySet является итерируемым, он выполняет свой запрос к базе данных при первом обращении.
- Slicing. Limit, offset.
- len(). Выполняет запрос и считает количество записей.
- list(). Принудительное выполнения путем вызова list() на QuerySet.
- bool(). Тестирование QuerySet в булевом контексте (bool (), or, and) приведет к выполнению запроса.

Перейдем к агрегации. Для более выразительных примеров добавим поле views в сущность entry.

```
class Entry(models.Model):
    blog = models.ForeignKey(Blog,
on_delete=models.CASCADE)
    headline =
models.CharField(max_length=255, blank=True,
null=True, default=None)
    views = models.IntegerField(default=0)
    authors = models.ManyToManyField(Author)
```

Рис. 42

В Django есть два понятия: агрегация и аннотация.

Рассмотрим агрегацию. Например, нам нужно посчитать количество блогов. Или нам нужно посчитать разницу между максимальным количеством просмотров `views` и его средним значением. Также перед `aggregate` можно выполнять различные фильтры, то есть `aggregate` можно вызывать у `QuerySet`. Можно сколько угодно раз вызывать до этого фильтр, потом сделать `aggregate` и для него уже посчитать некую агрегацию.

```
>>> Blog.objects.all().count()
>>> Blog.objects.all().aggregate(Count('pk'))
SELECT count("core_blog"."id") FROM "core_blog";

>>> Entry.objects.aggregate(diff=Max('views',
output_field=models.FloatField()) - Avg('views'))
SELECT max("core_entry"."views") -
avg("core_entry"."views")
FROM "core_entry";

>>>
Blog.objects.filter(name='name1').aggregate(Count('pk'))
SELECT count("core_blog"."id") FROM "core_blog" WHERE
name = name1;
```

Рис. 43

Перейдем к аннотации. То есть до этого мы агрегировали весь `QuerySet` и в результате получали одну какую-то цифру. Аннотацией мы будем аннотировать сущность. Например, нам нужно посчитать количество `entry` у каждого блога. То есть мы хотим получить все сущности `blog`, но добавить туда некий параметр, который будет соответствовать количеству `entry`.

```
>>> blogs = Blog.objects.annotate(Count('entry'))
>>> print(blogs.query)

SELECT "core_blog"."id", "core_blog"."name",
"core_blog"."tagline", COUNT("core_entry"."id") AS
"entry__count"
FROM "core_blog"
LEFT OUTER JOIN "core_entry" ON ("core_blog"."id" =
"core_entry"."blog_id") GROUP BY "core_blog"."id",
"core_blog"."name", "core_blog"."tagline"
```

Рис. 44

Тогда как aggregate можно было вызывать только после фильтра, а после aggregate нельзя было уже фильтр вызывать, после annotate можно вызывать фильтр. Например, нам нужно вывести все блоги, у которых количество entry > 0. Для этого мы после annotate вызываем фильтр по этому полю. Это так же, как и в aggregate, просто мы пофилтровали QuerySet. То есть нам нужно просто получить те сущности, у которых name=name и к ним добавить аннотацию количества entry.

```
>>> blogs =
Blog.objects.annotate(entry_count=Count('entry'))
.filter(entry_count__gt=0)

SELECT ... , COUNT("core_entry"."id") AS
"entry_count" FROM "core_blog"
LEFT OUTER JOIN "core_entry" ON (...)
GROUP BY ...
HAVING COUNT("core_entry"."id") > 0

>>> blogs =
Blog.objects.filter(name='name').annotate(entry_c
ount=Count('entry'))

SELECT ... , COUNT("core_entry"."id") AS
"entry_count" FROM "core_blog"
LEFT OUTER JOIN "core_entry" ON (...)
WHERE "core_blog"."name" = name
GROUP BY ...
```

Рис. 45

Также по аннотации можно сортировать, то есть вывести блоги в порядке убывания по количеству entry.

```
>>> blogs =
Blog.objects.annotate(entry_count=Count
('entry')).order_by('-entry_count')
>>> print(blogs.query)

SELECT ... , COUNT("core_entry"."id")
AS "entry_count" FROM "core_blog"
LEFT OUTER JOIN "core_entry" ON (...)
GROUP BY ...
ORDER BY "entry_count" DESC
```

Рис. 46

## 4. HTML

### 4.1. Панель разработчика в Chrome

Для того, чтобы открыть панель разработчика, необходимо нажать правую кнопку и в самом низу диалога, который выплывает, нажать inspect.

Давайте сначала начнем рассмотрение с панели Elements. В панели Elements представлен наше DOM дерево. Тут можно посмотреть наш HTML верстку, которое нам пришла. Если мы хотим выделить какой-то определенный элемент, нам это очень часто понадобится, когда мы будем верстать, нам необходимо нажать стрелку и навести на тот элемент, который мы хотим, например, на logo mail.ru. Мы видим, что эта ссылка, ведет она на mail.ru и мы можем увидеть, что с правой стороны и у нас окошко со стилями. Это те стили, которые будут применены к данному элементу.

Можно увидеть, что на этот элемент навешен стиль, который отвечает за logo, собственно, если его выключить, оно пропадет и таким образом можно отменять и добавлять стили. Если мы хотим проверить какой-то определенный стиль, например, display: none, мы увидим, что наш элемент пропал. Мы применили этот стиль к непосредственно этому элементу. Это очень удобно, когда мы верстаем и хотим попробовать, протестировать.

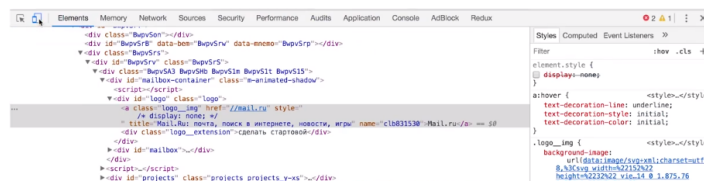


Рис. 47

Очень удобный инструмент, когда мы хотим проверить, как наш сайт будет выглядеть на мобильном устройстве. Нажимаем и у нас экран уменьшается в формат нашего мобильного устройства

и мы можем посмотреть, как же сайт будет выглядеть на нем. Можно выбрать то устройство, которое нам необходимо, он будет и менять user агент, и тут будет показано размер экрана и можно выбрать скорость интернета или вообще его выключить.

Следующий этап, который мы посмотрим, это network, тоже один из самых важных элементов в инспекторе. Тут мы можем видеть все, что происходит с сетью на нашей странице.

Очень важные элементы сверху, например, disable cache, это означает, что наш браузер не будет кэшировать статические файлы. Может быть такая проблема, что вы изменили в коде ваш файл и стиль не применился. Может быть это из-за того, что ваш браузер закэшировал этот файл и не получил обновления.

Давайте перейдем обратно в обычный режим. Можем видеть, что когда мы увеличили нашу страницу до размеров десктопа, то появились некоторые react запросы.

Давайте теперь перейдем на вкладку Application. Тут очень важный раздел storage. Это то, что хранит наши страницы, например, local storage, это локальное хранилище. Тут можно посмотреть все данные, которые хранит наша страница. И cookies storage, посмотреть те куки, которые установлены на наш домен mail.ru. Ими можно манипулировать, посмотреть на какой домен они устанавливаются, посмотреть время их жизни, посмотреть может ли они устанавливаться на запросе HTTP и так далее и тому подобное.

И вкладка console, это, грубо говоря, консольный вывод JS скриптов. Тут можно проверить некие наши скрипты, например,  $1+1$  и посмотреть некий вывод с помощью команды console.log.

## 4.2. Основы HTML

На картинке представлен простой HTML, его структура выглядит следующим образом.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Страница</title>
    <meta http-equiv="Content-Type"
      content="text/html"; charset="utf-8">
    <meta name="description" content="Сайт">
    <link rel="stylesheet" href="./style.css">
  </head>
  <body id="the_body">
    <p class="article"> ... </p>
    <script src="./script.js"></script>
  </body>
</html>
```

Рис. 48

Вначале идет DOCTYPE, далее идет тэг верхнего уровня HTML, и внутри него также тэги верхнего уровня head и body.

DOCTYPE уточняет тип содержимого, указывает HTML парсеру, как правильно разбирать данный документ. У HTML есть несколько стандартов, и например, DOCTYPE, который указан у



нас в примере, это HTML5.

Рассмотрим тэги верхнего уровня. HTML — это некая обертка, header — это заголовок страниц, которые не отображаются, и body — это тело страницы, которое видит пользователь.

Давайте теперь подробнее рассмотрим, что находится внутри тэга head. Это тэг title, который отображается в заголовке окна браузера, тэги meta — это некая метаинформация о нашей странице, link — это тэг, который указывает на связанные ресурсы, например, CSS. И script — там указываются наши файлы JavaScript. Загрузку CSS, это тэги link, рекомендуется ставить в тэг head, а загрузку JavaScript, наоборот, лучше ставить ближе к концу страницы, это повышает скорость отрисовки страницы.

Давайте теперь рассмотрим, что такое блочные элементы. Для начала рассмотрим примеры блочных элементов:

- h1 – h6 — различные уровни заголовков;
- p — разбиение текста на параграфы;
- hr — это горизонтальная линия;
- pre — это блок преформатированного кода;
- blockquote — это цитирование длинного блока текста;
- div — это некий абстрактный блочный контейнер.

Особенности блочных элементов:

- Блоки располагаются по вертикали друг под другом.
- На прилегающих сторонах элементов действует эффект схлопывания отступов. Например, если у нас указали отступ снизу у верхнего элемента, margin, и указали отступ сверху у нижнего элемента, и например, у верхнего элемента отступ 20 пикселей, у нижнего 10, то если они друг под другом, в итоге отступ все равно будет 20. Это есть эффект схлопывания.
- Запрещено вставлять блочный элемент внутрь строчного.
- По ширине блочный элемент занимает всё доступное пространство.
- Высота блочного элемента вычисляется браузером автоматически, исходя из его содержания.
- На блочные элементы не действуют свойства, предназначенные для строчных элементов, такие как vertical-align.

Теперь строчные элементы:

- а как гиперссылка;

- `i` — это акцентирование;
- `strong` и `b` — это выделение жирным текстом;
- `image` — это изображение;
- `sub` — это нижний индекс;
- `sup` — это верхний индекс;
- `span` — это некий абстрактный строчный контейнер.

Рассмотрим особенности строчных элементов:

- Внутри строчных элементов допустимо помещать текст и другие строчные элементы.
- Вставлять блочные элементы внутрь строчных нельзя.
- Эффект схлопывания отступов не действует.
- Свойства, связанные с размерами, не применимы.
- Ширина равна содержимому плюс значение отступов полей границ.
- Несколько строчных элементов, идущих подряд, располагаются на одной строке и переносятся на другую только при необходимости.
- Можно выравнивать по вертикали с помощью `vertical-align`.

Давайте теперь рассмотрим стандартные HTML-элементы.

Начнем со списков. В HTML есть маркированные списки и нумерованные списки. Маркированные списки — это `ul`, нумерованные — это `ol`. Для того чтобы, например, сделать маркированный список, нам сначала нужно написать тэг `ul` и внутри него перечислить все пункты с помощью тэга `li`.

Таблица. Таблица в HTML делается с помощью тэга `table`. Если необходимо сделать заголовок, это делается с помощью тэга `caption`. Далее определяется заголовок таблицы с помощью тэга `thead` и там формируется этот заголовок с помощью тэга `tr` и тэгов `td`. Далее идет тело нашей таблицы, оно формируется с помощью тэга `tbody` и также с помощью тэгов `tr` и `td`.

```
<table border="1">
  <caption>квартальный отчет </caption>
  <thead>
    <tr>
      <td>дата</td>
      <td colspan="2">доход</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th rowspan="2">2011-01-01</th>
      <td>100500</td>
      <td>33</td>
    </tr>
    <tr>
      <td>100</td>
      <td>500</td>
    </tr>
  </tbody>
</table>
```

квартальный отчет

дата	доход	
2011-01-01	100500	33
	100	500

Рис. 49

Теперь давайте рассмотрим гиперссылки. Для того чтобы сделать ссылку, необходимо использовать тэг `a`, и рассмотрим параметры, которые можно передать внутри. Это `href` — это url к гиперссылке, `target` — это в каком окне открывать ссылку, `name` — это имя якоря вместо `href`. Внутри `a` можно указывать тэги или просто текст. Действия браузера при переходе по ссылке. Мы не только можем перейти по ссылке и открыть какой-то другой веб-ресурс, а можно выполнить какое-то другое действие. Например, если мы в схеме укажем `http`, `https`, `ftp`, это будет переход по ссылке, если укажем `mailto`, то он запустит почтовый клиент. Если указать `javascript`, будет выполнен JavaScript-код, если указать внутри ссылки якорь, то есть `anchor`, то произойдет прокрутка до ID, которой соответствует этот якорь.

Для того чтобы вставить изображение, нужно использовать тэг `img` и указать там параметры, такие как `src` — это путь к нашему изображению, `alt` — это альтернативный текст для изображения, он появится при наведении и ожидании некоторого времени курсора на этой картинке. И указание размеров с помощью `width` и `height`.

```

```