

Лекция 3. Работа с представлениями и шаблонам



На этой лекции мы

1. Узнаем о представлениях Django
2. Разберемся в работе диспетчера URL
3. Изучим шаблоны и передачу контекста в них
4. Узнаем о условиях, циклах и наследовании шаблонов
5. Объединяем модели, представления, шаблоны и маршруты

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Узнали о моделях Django
2. Разобрались в создании моделей
3. Изучили миграции
4. Узнали о создании собственных команд
5. Изучили работу с моделями данных, CRUD

План лекции

[На этой лекции мы](#)

[Краткая выжимка, о чём говорилось в предыдущей лекции](#)

[План лекции](#)

[Подробный текст лекции](#)

[Представления](#)

[Вместо старта](#)

[Представления на основе функций](#)

[Представления на основе классов](#)

[Диспетчер URL](#)

[Сопоставление представления с маршрутом](#)

[Передача параметров](#)

[Преобразование пути в типы Python](#)

[Из URL во view](#)

[Представление через функцию возвращает HttpResponse](#)

[Представление через класс возвращает HttpResponse](#)

[Представление через функцию возвращает JSON](#)

[Шаблоны](#)

[Каталог шаблона](#)

[Первый шаблон](#)

[Передача контекста в шаблон](#)

[Проверка условия в шаблонах](#)

[Пробрасываем контекст в if шаблон используя TemplateView](#)

[Добавление маршрута](#)

[Вывод в цикле](#)

[Наследование шаблонов Django](#)

[Базовый шаблон проекта](#)

[Объединяем модели, представления, шаблоны и маршруты](#)

[Модели](#)

[Создание моделей](#)

[Миграции](#)

[Наполнение фейковыми данными](#)

[Представления](#)

[Представление автора](#)

[Представление статьи](#)

[Маршруты](#)

[Шаблоны](#)

[Базовый шаблон](#)

[Шаблон с последними статьями автора](#)

[Шаблон статьи](#)

[Вывод](#)

[Домашнее задание](#)

Подробный текст лекции

Представления

Представление в Django — это функция или класс, которая обрабатывает запрос и возвращает ответ в виде HTTP-ответа. Оно определяет, какие данные будут отображаться на странице и как они будут отображаться.

Представления располагаются в файле `views.py` вашего приложения. Если проект состоит из нескольких приложений, каждое будет иметь свои “вьюшки” в собственном каталоге.

Вместо старта

Если в рамках урока вы создаёте новое приложение, выполните команды:

```
>cd myproject
>python manage.py startapp myapp3
```

Сразу добавьте приложение в константу со списком приложений

```
INSTALLED_APPS = [
    ...
    'myapp2',
    'myapp3',
]
```

Представления на основе функций

Функциональное представление — это функция Python, которая принимает объект запроса и возвращает объект ответа. Она может быть определена как обычная функция или декоратор. Пример функционального представления:

```
from django.http import HttpResponse
```

```
def hello(request):  
    return HttpResponse("Hello World from function!")
```

В этом примере мы импортируем класс `HttpResponse` из модуля `django.http` и определяем функцию `hello`, которая принимает объект запроса `request` и возвращает объект ответа `HttpResponse` с текстом "Hello, World!".

Представления на основе классов

Классовое представление – это класс Python, который наследуется от базового класса `View` и реализует один или несколько методов для обработки запросов. Пример классового представления:

```
from django.views import View  
from django.http import HttpResponse  
  
class HelloView(View):  
    def get(self, request):  
        return HttpResponse("Hello World from class!")
```

В этом примере мы импортируем базовый класс `View` из модуля `django.views` и определяем класс `HelloView`, который наследуется от него. Метод `get()` класса `HelloView` обрабатывает GET-запросы и возвращает объект ответа `HttpResponse` с текстом "Hello, World!".

В Django существует множество других типов представлений, таких как шаблонные представления, которые используют шаблоны HTML для отображения данных, или API-представления, которые возвращают данные в формате JSON или XML. Однако функциональные и классовые представления являются наиболее распространенными и простыми в использовании.

Далее в рамках лекции будем использовать различные варианты представлений, чтобы на примерах закрепить навыки по их созданию. В реальных проектах стоит выбрать единую концепцию для всех представлений вашего проекта.

Диспетчер URL

Диспетчер URL является одним из ключевых компонентов фреймворка Django, который отвечает за обработку входящих запросов и направление их на соответствующие обработчики.

Обработка запросов в Django осуществляется следующим образом: когда пользователь делает запрос к веб-приложению, сервер Django получает этот запрос и передает его диспетчеру URL. Диспетчер URL анализирует запрос и определяет, какой обработчик должен быть вызван для его обработки.

Сопоставление представления с маршрутом

Использование представлений осуществляется путем указания их имени в URL-адресе приложения. Например, для функционального представления `hello()` и классового представления `HelloView` можно определить URL-шаблоны следующим образом. Создаём файл `urls.py` в каталоге приложения и пишем код:

```
from django.urls import path
from .views import hello, HelloView

urlpatterns = [
    path('hello/', hello, name='hello'),
    path('hello2/', HelloView.as_view(), name='hello2'),
]
```

Здесь мы определяем URL-шаблоны для функционального представления `hello()` и классового представления `HelloView`. В первом случае мы указываем имя функции в качестве обработчика запроса, во втором – используем метод `as_view()` класса `HelloView` для создания объекта-обработчика запроса.

Далее нам надо подключить маршруты приложения к маршрутам проекта. Открываем `urls.py` в каталоге проекта и вносим в него изменения. Получим примерно следующий код:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
```

```
path('admin/', admin.site.urls),
...
path('les3/', include('myapp3.urls')),
]
```

Добавляется маршрут для приложения `myapp3`, который будет обрабатывать URL-адрес `les3` в качестве префикса. В случае его совпадения передавать управление в модуль `myapp3.urls`. Маршрут включается с помощью функции `include`, а дальнейшая обработка адреса происходит в `urls.py` приложения. Его мы создали парой абзацев выше.

Передача параметров

Преобразования пути — это процесс преобразования URL-адреса запроса в формат, понятный для Django. Django преобразует пути запроса в параметры, которые передаются обработчику. Рассмотрим пример кода в файле `myapp3/urls.py`:

```
from django.urls import path
...
from .views import year_post, MonthPost, post_detail

urlpatterns = [
    ...
    path('posts/<int:year>/', year_post, name='year_post'),
    path('posts/<int:year>/<int:month>/', MonthPost.as_view(), name='month_post'),
    path('posts/<int:year>/<int:month>/<slug:slug>/', post_detail, name='post_detail'),
]
```

В этом примере мы определяем три маршрута для обработки запросов. В первом маршруте мы используем параметр `<int:year>`, чтобы указать год статьи. Во втором маршруте мы используем параметры `<int:year>` и `<int:month>`, чтобы указать год и месяц статьи. В третьем маршруте мы используем параметры `<int:year>`, `<int:month>` и `<slug:slug>`, чтобы указать год, месяц и уникальный идентификатор статьи.

Когда пользователь делает запрос к веб-приложению, Django преобразует URL-адреса запроса в параметры, которые передаются обработчику. Например,

если пользователь запрашивает статью, опубликованную в июне 2022 года с идентификатором "python", Django преобразует URL-адрес запроса в следующие параметры:

```
{
    'year': 2022,
    'month': 6,
    'slug': 'python',
}
```

Эти параметры затем передаются обработчику `views.post_detail`, который может использовать их для отображения соответствующей статьи.

Таким образом, диспетчер URL и преобразования пути являются важными компонентами фреймворка Django, которые позволяют обрабатывать запросы и направлять их на соответствующие обработчики.

Преобразование пути в типы Python

В Django преобразование путей осуществляется с помощью приставок, которые определяют тип данных, который будет передаваться в качестве параметра в представление. Для этого мы заключаем параметр в треугольные скобки и указываем приставку, а далее после двоеточия слитно пишем имя параметра.

- `str` — приставка для передачи строки любых символов, кроме слэша. Например, если мы хотим передать в представление информацию о конкретном посте блога, то мы можем использовать такой путь:
`path('posts/<str:slug>/', post_detail)`. Здесь `slug` - это строка символов, которая является уникальным идентификатором поста.
- `int` — приставка для передачи целого числа. Например, если мы хотим передать в представление информацию о конкретном пользователе по его идентификатору, то мы можем использовать такой путь:
`path('users/<int:id>/', user_detail)`. Здесь `id` - это целое число, которое является уникальным идентификатором пользователя.
- `slug` — приставка для передачи строки, содержащей только буквы, цифры, дефисы и знаки подчеркивания. Например, если мы хотим передать в представление информацию о конкретной категории товаров, то мы можем использовать такой путь:
`path('categories/<slug:slug>/', category_detail)`. Здесь `slug` - это строка символов, которая является уникальным идентификатором категории.
- `uuid` — приставка для передачи уникального идентификатора. Например, если мы хотим передать в представление информацию о конкретном заказе, то мы можем использовать такой путь:

`path('orders/<uuid:pk>/', order_detail)`. Здесь `pk` - это уникальный идентификатор заказа.

- `path` — приставка для передачи строки любых символов, включая слэши. Например, если мы хотим передать в представление информацию о конкретном файле на сервере, то мы можем использовать такой путь:

`path('files/<path:url>/', file_detail)`. Здесь `url` - это строка символов, которая содержит путь к файлу на сервере.

Из URL во view

Снова возвращаемся к представлениям. Следующая строка кода из `myapp3/urls.py` в настоящий момент вызывает ошибки импорта:

```
from .views import year_post, MonthPost, post_detail
```


Рассмотрим простейшие вьюшки, которые устранят ошибки в нашем приложении.

Представление через функцию возвращает `HttpResponse`

```
from django.http import HttpResponse

def year_post(request, year):
    text = ""
    ... # формируем статьи за год
    return HttpResponse(f"Posts from {year}<br>{text}")
```

Это представление будет доступно по адресу <http://127.0.0.1:8000/les3/posts/2022/>. При этом год может быть любым целым числом.

 **Внимание!** Если вы указали другой префикс в корневом `urls`, ваш адрес будет отличаться.

Представление через класс возвращает `HttpResponse`

```
from django.views import View
from django.http import HttpResponse

class MonthPost(View):
    def get(self, request, year, month):
```

```

text = ""
... # формируем статьи за год и месяц
return HttpResponse(f"Posts from
{month}/{year}<br>{text}")

```

Вторая вьюшка основана на классе. Работает метод аналогично первой функции. Но в качестве параметров мы получаем и год и месяц. Например можно перейти по адресу <http://127.0.0.1:8000/les3/posts/2022/6/>

Представление через функцию возвращает JSON

```

from django.http import JsonResponse

def post_detail(request, year, month, slug):
    ... # Формируем статьи за год и месяц по идентификатору.
    Пока обойдёмся без запросов к базе данных
    post = {
        "year": year,
        "month": month,
        "slug": slug,
        "title": "Кто быстрее создаёт списки в Python, list() или
[]",
        "content": "В процессе написания очередной программы
задумался над тем, какой способ создания списков в Python
работает быстрее..."
    }
    return JsonResponse(post, json_dumps_params={'ensure_ascii':
False})

```

В отличие от двух первых представлений, третье возвращает JSON объект. Очевидное изменение — использование JsonResponse вместо привычного HttpResponse. Менее очевидное - русский текст. А если быть более точным, текст в кодировке UTF-8, а не в ASCII. Для этого мы передаём дополнительный параметр `json_dumps_params={'ensure_ascii': False}`. Если вы работали с модулем json из стандартной библиотеки Python, параметр `ensure_ascii` вам знаком. Он подтверждает, что JSON будет содержать не только 127 символов из кодировки ASCII.

Проверить работу представления можно по адресу наподобие <http://127.0.0.1:8000/les3/posts/2022/6/python/>

Шаблоны

Шаблоны в Django представляют собой файлы, содержащие HTML-код с дополнительными тегами и переменными, которые могут быть заменены на значения из контекста.

Шаблоны в Django используются для генерации HTML-страниц на основе данных, полученных из представлений. Они позволяют создавать динамические страницы, которые могут меняться в зависимости от данных, полученных от пользователя или из базы данных.


Каталог шаблона

Прежде чем создавать первый шаблон, разберёмся в правильной структуре каталогов.

Внутри каталога приложения необходимо создать каталог `templates`. Далее в нём создаётся каталог с именем приложения. Схема с двумя приложениями в проекте ниже:

```
myproject/
  myapp1/
    templates/
      myapp1/
        index.html
        ...
      ...
  myapp2/
    templates/
      myapp2/
        index.html
        ...
      ...
myproject/
  ...
...
manage.py
```

В каждом приложении есть каталог `templates`. Если бы мы не создали внутри него каталог с именем приложения (`myapp1` и `myapp2`), Python и Django не смогли бы различать шаблоны `index.html` разных приложений.


 **Важно!** На первый взгляд сложная структура каталогов приложение/служебный_каталог/приложение в действительности упрощает разработку и переносимость кода между проектами. Стоит сразу привыкать к ней, придерживаться во всех проектах, даже учебных.

Первый шаблон

Давайте рассмотрим пример создания простого шаблона. Назовём его my_template.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Первый шаблон Django</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

В этом примере мы создали HTML-страницу с заголовком "Первый шаблон Django". Внутри тега body мы добавили заголовок первого уровня h1 и использовали переменную name, которая будет передана в шаблон из представления.

 **Внимание!** Переменные внутри шаблона заключаются в двойные фигурные скобки с пробелами до и после имени переменной.

Передача контекста в шаблон

Чтобы передать данные в шаблон, мы можем использовать функцию render из модуля django.shortcuts:

```
from django.shortcuts import render

def my_view(request):
```

```
context = {"name": "John"}
return render(request, "myapp3/my_template.html", context)
```

В этом примере мы создали функцию `my_view`, которая использует шаблон `my_template.html` из приложения `myapp3`. Функция передает шаблону параметр `name` со значением "John". Функция `render` заменяет переменные в шаблоне на значения из контекста и возвращает готовую HTML-страницу.

Не забываем про регистрацию нового представления в `urls.py` приложения.

```
...
from .views import my_view

urlpatterns = [
    ...
    path('', my_view, name='index'),
]
```

Отлично! Можно перейти по адресу <http://127.0.0.1:8000/les3/> для проверки работы шаблона.

Проверка условия в шаблонах

В Django Framework условные операторы в шаблонах имеют синтаксис похожий на Python и заключаются в фигурные скобки вида `{% %}`. Обязательным является закрывающий оператор кода вида `{% endif %}`.

Пример использования условного оператора в шаблоне Django `templ_if.html`:

```
{% if message %}
    <p>Вам доступно сообщение: <br> {{ message }}</p>
{% endif %}
```

В данном примере, если в шаблон передали переменную `message`, будет выведен абзац текста. В противном случае код между открывающим и закрывающим операторами будет проигнорирован и не появится на стороне клиента.

Кроме того, шаблоны Django поддерживают сложные условия благодаря конструкциям `{% elif %}` и `{% else %}`. Например, мы можем выбирать окончание предложения в зависимости от переданного числа:

```

<p>К прочтению предлагается {{ number }}
    {% if number == 1 %}
        пост
    {% elif number >= 2 and number <= 4 %}
        поста
    {% else %}
        постов
    {% endif %}
</p>

```

В данном примере, в зависимости от значения переменной `number` будет выбрано соответствующее окончание предложения. Если `number` равно 1, будет выведено "пост", если от 2 до 4 - "поста", иначе - "постов".

Важно помнить, что перед использованием переменной в условии, ее необходимо передать в шаблон через словарь `context`.

Пробрасываем контекст в if шаблон используя TemplateView

Чтобы увидеть HTML страницу на основе шаблона `templ_if.html` надо пробросить в него контекст. Рассмотрим ещё одну вариацию представления.

```

from django.views.generic import TemplateView

...
class TemplIf(TemplateView):
    template_name = "myapp3/templ_if.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['message'] = "Привет, мир!"
        context['number'] = 5
        return context

```

В данном примере мы создаем класс `TemplIf`, который наследуется от `TemplateView` и указывает на использование шаблона `myapp3/templ_if.html`. В методе `get_context_data` мы добавляем две переменные в словарь `context` - `message` и `number`. В шаблоне мы используем эти переменные в условных операторах. Если переменная `message` не пуста, будет выведено сообщение. В зависимости от значения переменной `number` будет выбрано соответствующее окончание предложения.

Добавление маршрута

Чтобы получить доступ к вьюшке, а через неё к шаблону, надо внести дополнение в `urls.py`

```
from django.urls import path
from .views import HelloView, TemplIf
...

urlpatterns = [
    ...
    path('if/', TemplIf.as_view(), name='templ_if'),
]
```

Стандартное добавление пути с использованием метода `as_view` для класса.

Вывод в цикле

В Django для вывода данных в цикле используется тег `for`. Он позволяет перебирать элементы списка, словаря или `QuerySet`'а и выводить их на страницу. Примеры кода шаблона с циклом (файл `templ_for.html`):

1. Вывод элементов списка:

```
<h2>Элементы списка</h2>
<ul>
{% for item in my_list %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

2. Вывод ключей и значений словаря в таблицу:

```
<h2>Ключи и значения словаря</h2>
<table>
{% for key, value in my_dict.items %}
    <tr><td>{{ key }}</td><td>{{ value }}</td></tr>
{% endfor %}
</table>
```



Внимание! Как и для условия if, цикл for обязан завершаться оператором
{% endfor %}

А для работы шаблона напишем код представления на основе функции с передачей списка и словаря в контексте:

```
from django.shortcuts import render

...

def view_for(request):
    my_list = ['apple', 'banana', 'orange']
    my_dict = {
        'каждый': 'красный',
        'охотник': 'оранжевый',
        'желает': 'жёлтый',
        'знать': 'зелёный',
        'где': 'голубой',
        'сидит': 'синий',
        'фазан': 'фиолетовый',
    }
    context = {'my_list': my_list, 'my_dict': my_dict}
    return render(request, 'myapp3/templ_for.html', context)
```

В данном примере мы создаем функцию view_for, которая передает список my_list и словарь my_dict в контекст шаблона и вызывает рендеринг шаблона myapp3/templ_for.html. В шаблоне мы можем использовать тег for для вывода элементов списка.

Не забываем про импорт представления в urls.py и добавление строки маршрута:

```
from django.urls import path
from .views import view_for

...

urlpatterns = [
    ...
    path('for/', view_for, name='templ_for'),
]
```


Работа кода шаблона заключается в том, что он перебирает элементы списка или словаря с помощью тега `for` и выводит их на страницу с помощью переменных, которые обернуты в двойные фигурные скобки. Код представления передает данные в контекст шаблона, которые затем используются в шаблоне для вывода на страницу.

Наследование шаблонов Django

В Django есть возможность наследовать шаблоны, что позволяет создавать базовый шаблон и на его основе создавать другие, которые будут иметь общий вид и функциональность. Наследование шаблонов позволяет избежать дублирования кода и упростить процесс разработки. Мы перестаём нарушать принцип DRY.

Рассмотрим примеры двух шаблонов без наследования.

Шаблон 1:

```
<!DOCTYPE html>
<html>
<head>
    <title>Мой сайт</title>
</head>
<body>
    <h1>Приветствую на моём сайте!</h1>
    <p>Это главная страница.</p>
</body>
</html>
```

Шаблон 2:

```
<!DOCTYPE html>
<html>
<head>
    <title>Обо мне</title>
</head>
<body>
    <h1>Обо мне</h1>
    <p>Меня зовут Алексей и я пишу программы с 14 лет.</p>
</body>
</html>
```

Оба шаблона имеют повторяющийся код, например, теги DOCTYPE, html, head и body. Чтобы избежать дублирования кода, можно использовать наследование.

Пример базового шаблона с наследованием:

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Сайт{% endblock %}</title>
</head>
<body>
  {% block content %}
    <p>Скоро тут появится текст...</p>
  {% endblock %}
</body>
</html>
```

В этом примере мы создали базовый шаблон, который содержит блоки title и content. Текст внутри блоков является резервным вариантом. Если дочерний шаблон не переопределит блок, будет выведен текст внутри блоков базового шаблона. Блоки позволяют переопределять содержимое в наследуемых шаблонах.

Пример шаблона, наследующего базовый:

```
{% extends 'myapp3/base.html' %}

{% block title %}Обо мне{% endblock %}

{% block content %}
  <h1>Обо мне</h1>
  <p>Меня зовут Алексей и я пишу программы с 14 лет.</p>
{% endblock %}
```

В этом примере мы использовали тег extends для указания базового шаблона. Затем мы переопределили блок title и добавили содержимое в блок content.



Важно! Если базовый шаблон находится во внутреннем каталоге приложения, первая строка будет следующей:

```
{% extends 'app/base.html' %}
```

Теги `block` и `extends` позволяют создавать базовые шаблоны и наследовать их в других шаблонах. Блоки позволяют переопределять содержимое в наследуемых шаблонах, а тег `extends` указывает, какой шаблон является базовым для текущего.



Внимание! Как и во всех прошлых случаях вам нужно создать представления, которые будут отрисовывать шаблоны, а также добавить маршруты для представлений в `urls.py`

Базовый шаблон проекта

Django позволяет создать базовый шаблон на уровне проекта. В таком случае все приложения смогут использовать его для расширения своих дочерних шаблонов.

Для этого выполним предварительные настройки в файле `settings.py` проекта:

```
TEMPLATES = [
    {
        # 'BACKEND' is the template engine. Default is 'django.template.backends.django.DjangoTemplates',
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            # Add the project root path as the first element
            BASE_DIR / 'templates',
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            # List of context processors to use, either as a list of strings, or as a list of
            # (string, callable) pairs.
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

Добавляем в список `DIRS` путь до каталога шаблона проекта - `BASE_DIR / 'templates'`.

Далее создаём каталог `templates` в каталоге `BASE_DIR`. Это каталог верхнего уровня. В нём находится файл `manage.py`, каталог проекта и каталоги приложений

```
myproject/
    myapp1/
```

```
...
myapp2/
...
myproject/
...
templates/
    base.html
...
manage.py
```

В каталог помещаем базовый шаблон приложения base.html.

Теперь команда расширения в дочерних шаблонах будет записываться без указания имени приложения:

```
{% extends 'base.html' %}
```

Объединяем модели, представления, шаблоны и маршруты

В финале соединим полученные на этой и прошлых лекциях знания в едином приложении. У нас будет база данных с авторами и постами (создавали на занятии про модели). Пользователь сможет вводить в адресной строке id автора и получать информацию о его пяти последних статьях. Также можно получить полный текст статьи по её id. URL маршруты вызывают соответствующие представления, которые в свою очередь обращаются через модель к базе данных и передают её через контекст в шаблоны.

Модели

Создание моделей

В файл models.py перенесём ранее созданный код:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
```

```

email = models.EmailField()

def __str__(self):
    return f'Name: {self.name}, email: {self.email}'

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

    def __str__(self):
        return f'Title is {self.title}'

    def get_summary(self):
        words = self.content.split()
        return f'{" ".join(words[:12])}...'

```

Автор имеет имя и почту. Статья состоит из заголовка и содержимого. При этом каждая статья имеет одного автора, а автор может писать множество статей.

Миграции

Создадим миграции:

```
python manage.py makemigrations myapp3
```

Сразу применением их к базе данных:

```
python manage.py migrate
```

Наполнение фейковыми данными

Чтобы было что выводить, заполним таблицы фейковыми данными. Для этого создадим файл myapp3/management/commands/fill_db.py:

```

from random import choices

from django.core.management.base import BaseCommand
from myapp3.models import Author, Post

LOREM = "Lorem ipsum dolor sit amet, consectetur adipisicing  
elit. " \
        "Accusamus accusantium aut beatae consequatur  
consequuntur cumque, delectus et illo iste maxime " \

```

```

        "nihil non nostrum odio officia, perferendis placeat
        quasi quibusdam quisquam quod sunt " \
        "tempore temporibus ut voluptatum? A aliquam culpa
        ducimus, eaque eum illo mollitia nemo " \
        "tempore unde vero! Blanditiis deleniti ex hic,
        laboriosam maiores odit officia praesentium " \
        "quae quisquam ratione, reiciendis, veniam. Accusantium
        assumenda consectetur consequatur " \
        "consequuntur corporis dignissimos ducimus eius est eum
        expedita illo in, inventore " \
        "ipsum iusto maiores minus mollitia necessitatibus neque
        nisi optio quasi quo quod, " \
        "quos rem repellendus temporibus totam unde vel velit
        vero vitae voluptates."

```

```

class Command(BaseCommand):
    help = "Generate fake authors and posts."

    def add_arguments(self, parser):
        parser.add_argument('count', type=int, help='User ID')

    def handle(self, *args, **kwargs):
        text = LOREM.split()
        count = kwargs.get('count')
        for i in range(1, count + 1):
            author = Author(name=f'Author_{i}',
email=f'mail{i}@mail.ru')
            author.save()
            for j in range(1, count + 1):
                post = Post(
                    title=f'Title-{j}',
                    content=" ".join(choices(text, k=64)),
                    author=author
                )
                post.save()

```

Для заполнения базы семью авторами необходимо выполнить команду

```
python manage.py fill_db 7
```

Отлично! Модели созданы, таблицы в БД существуют и заполнены данными.

Представления

Представление автора

Создадим “вьюшку” для получения 5 последних статей автора:

```
from django.shortcuts import render, get_object_or_404
from .models import Author, Post

def author_posts(request, author_id):
    author = get_object_or_404(Author, pk=author_id)
    posts = Post.objects.filter(author=author).order_by('-id')[:5]
    return render(request, 'myapp3/author_posts.html', {'author': author, 'posts': posts})
```

Новая функция `get_object_or_404` работает аналогично `get`, т.е. делает `select` запрос к базе данных. Но если запрос не вернёт строку из таблицы БД, представление отрисует страницу с ошибкой 404.

Также обратите внимание на метод `order_by('-id')`. После фильтрации статей по автору, мы сортируем их на основе `id` по убыванию. Об этом говорит знак минус перед именем. Далее питоновский срез формирует список из пяти статей с максимальными идентификаторами.

Словарь с контекстом в виде автора и списка статей пробрасываются в шаблон `myapp3/author_posts.html`.

Представление статьи

Второе представление должно возвращать шаблон с полным текстом статьи:

```
def post_full(request, post_id):
    post = get_object_or_404(Post, pk=post_id)
    return render(request, 'myapp3/post_full.html', {'post': post})
```

Сделав `select` запрос к таблице с постами мы передаём в шаблон `myapp3/post_full.html` контекст в виде одной статьи.

Маршруты

Сразу пропишем маршруты для вновь созданных представлений в файле `urls.py`:

```
from django.urls import path
...
from .views import author_posts, post_full

urlpatterns = [
    ...
    path('author/<int:author_id>/',      author_posts,
name='author_posts'),
    path('post/<int:post_id>/', post_full, name='post_full'),
]
```


Мы создаем два URL-адреса - для представлений `author_posts` и `post_full`. В обоих случаях мы используем целочисленный параметр в URL для передачи `id` автора и поста соответственно. Мы также добавляем имена для каждого URL-адреса, чтобы мы могли ссылаться на них в шаблонах с помощью тега `url`. О теге `url` вы узнаете через несколько абзацев.

Шаблоны

Базовый шаблон

Начнём с простейшего базового шаблона. При желании добавить шапку и подвал в будущем, правки нужно делать только в нём. Создадим `base.html`:

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Блог{% endblock %}</title>
</head>
<body>
    {% block content %}
        Контент скоро появится...
    {% endblock %}
</body>
</html>
```


 **Внимание!** Расположите базовый шаблон в каталоге templates проекта. Дочерние шаблоны сохраняйте в каталоге templates/myapp3/ приложения.

Шаблон с последними статьями автора

Теперь подключим его в дочернем шаблоне author_posts.html для вывода 5 последних статей автора:

```
{% extends 'base.html' %}

{% block title %}{{ author.name }}'s Posts{% endblock %}

{% block content %}
    <h2>Последние 5 статей автора: {{ author.name }}</h2>
    <table>
        {% for post in posts %}
            <tr>
                <td><a href="{% url 'post_full' post.id %}">{{
post.title }}</a>
                <td>{{ post.get_summary }}</td>
            </tr>
        {% endfor %}
    </table>
{% endblock %}
```

Внутри шаблона мы обращаемся к переменным author и post как к экземплярам класса. Для получения имени автора используем точечную нотацию author.name, т.к. это свойство прописано в модели автора. Аналогично получаем заголовок статьи.

Кроме того в модели Post есть метод get_summary. Он возвращает 12 первых слов из содержимого статьи. Используя {{ post.get_summary }} шаблон через контекст вызывает метод модели.

Отдельного внимания заслуживает тег url. После него в кавычках мы указываем имя представления, которое хотим вызвать. Внимательно посмотрите на строку из urls.py

```
path('post/<int:post_id>/', post_full, name='post_full'),
```

Мы можем обращаться к post_full по имени, потому что прописали ключевой аргумент name в функции path.

Передача значения `post.id` позволяет задать значение параметра `post_id` внутри представления `post_full`.

Шаблон статьи

Финальный штрих — шаблон для вывода полного текста статьи.

```
{% extends 'base.html' %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
    <h3>{{ post.title }}</h3>
    <p>{{ post.content }}</p>
{% endblock %}
```

В шаблоне мы также переопределили блок `title` и добавили содержимое в блок `content`. Выводим заголовок и полный текст поста.

Вывод

На этой лекции мы:

1. Узнали о представлениях Django
2. Разобрались в работе диспетчера URL
3. Изучили шаблоны и передачу контекста в них
4. Узнали о условиях, циклах и наследовании шаблонов
5. Объединили модели, представления, шаблоны и маршруты

Домашнее задание

1. Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.
2. *Загляните в официальную документацию Django и изучите дополнительные возможности работы со статическими файлами.