



Урок 2

Шаблон + Контекст = html

Шаблонизатор Django (теги, фильтры и наследование). Работа со статикой и ссылками на страницах. Отправка контента в шаблоны и загрузка его в контроллеры из внешних источников.

[Шаблоны в Django](#)

[Правильная работа со статикой в шаблонах](#)

[Динамические url-ссылки в шаблонах](#)

[Наследование шаблонов в Django](#)

[Подключение подшаблонов в Django](#)

[Передача контекста в шаблон](#)

[Закрепление материала](#)

[*Работа с меню](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Шаблоны в Django

На прошлом уроке мы запустили наш проект и добились определенных успехов: происходит отображение и переход между тремя полноценными статическими html-страницами.

Будучи веб-фреймверком, Django позволяет динамически генерировать HTML. Самый распространенный подход — использование шаблонов. Шаблоны содержат статический HTML и динамические данные, рендеринг которых описан специальным синтаксисом. Эти данные и называются **контекстом (context)**.

За работу с различными компонентами фреймворка отвечают специальные модули — бэкенды (backend). Благодаря такому подходу достигается гибкость в конфигурировании и достижении требуемого функционала.

По умолчанию в django-проекте в константе `TEMPLATES` прописано:

```
'BACKEND': 'django.template.backends.django.DjangoTemplates'
```

Задача бэкенда — загрузка шаблона и обработка с учетом контекста. В результате получается готовая html-страница, которую сервер и возвращает пользователю.

Замечание: при необходимости можно подключить другой backend — например, 'Jinja2'. Мы не будем менять настройку и оставим 'DjangoTemplates'.

Для подстановки динамических данных в шаблонизаторе используются так называемые **плейсхолдеры (placeholders)**:

```
{{ имя переменной из контроллера }}
```

Если в Python мы для передачи **индекса** или имени **ключа** используем квадратные скобки:

```
user['first_name'],
```

то в шаблонизаторе будем **всегда** использовать точку:

```
{{ user.first_name }}
```

В языке шаблонов также есть аналоги операторов из Python. Они называются шаблонными **тегами** и записываются в виде:

```
{% имя тега %}
```

Например, вывод текущей даты в шаблоне можно сделать при помощи тега:

```
{% now "Y" %}
```

Цикл в шаблонах Django записывается тоже через теги:

```
{% for item in items_list %}  
    <p>{{ item }}</p>  
{% endfor %}
```

Нужно некоторое время, чтобы привыкнуть к **закрывающим тегам** (в данном примере — `endfor`).

Пример условного оператора:

```
{% if user.first_name %}  
    <p>{{ user.first_name }}</p>  
{% else %}  
    <p>Неизвестный</p>  
{% endif %}
```

Когда мы используем имена переменных внутри тегов, ставить дополнительные скобки не нужно:

```
{% if {{ user.first_name }} %} - будет ошибка!
```

Также язык шаблонизатора Django позволяет нам делать некоторые преобразования над переменными (хотя по идее вся обработка должна происходить на языке Python в файлах `views.py`). Эти преобразования называются **шаблонными фильтрами**.

Сначала может показаться непривычно.

По сути, шаблонный фильтр — это просто функция, написанная на Python, которая возвращает измененные данные в точку вызова. В конце курса мы напишем собственные шаблонные фильтры.

Пример шаблонного фильтра `title`, преобразующего первую букву слова в заглавную:

```
{{ user.first_name|title }}
```

Фильтр пишется через вертикальную черту после переменной и может принимать дополнительные аргументы через двоеточие справа:

```
{{ user.date_birth|date:"d m Y" }}
```

Замечание: будьте внимательны при работе с пробелами, когда используете шаблонные фильтры или теги. Иногда лишний пробел или его отсутствие — может привести к ошибке.

Обязательно почитайте [оригинальную документацию \(english\)](#) и посмотрите примеры. Названия почти всех тегов и фильтров интуитивно понятны и похожи на язык Python.

Дальше будем двигаться в двух направлениях:

- совершенствовать шаблоны;
- наполнять их динамикой.

Правильная работа со статикой в шаблонах

На прошлом уроке мы прописали в шаблонах для всех статических файлов общий url-корень: `'/static/'`. Если понадобится изменить этот адрес, то во всех шаблонах придется делать правки — высока вероятность ошибок. В Django существует специальный механизм, позволяющий решить эту проблему. Мы уже знакомы с константой `STATIC_URL`, значение которой по умолчанию `'/static/'` (файл `settings.py`). Пропишем в начале шаблона тег:

```
{% load staticfiles %}
```

Теперь можно ссылки на статические файлы оформить следующим образом:

```
<link rel="stylesheet" href="{% static 'css/style.css' %}">
```

Для сравнения – раньше было прописано:

```
<link rel="stylesheet" href="/static/css/style.css">
```

По сути, через тег `{% static '<относительный_путь>' %}` Django пробрасывает в шаблон значение `STATIC_URL` и соединяет с относительным путем к файлу. Если, например, мы пропишем значение `STATIC_URL='/newstatic/'`, то статические файлы будут загружаться с url-адреса `'/newstatic/'`.

Замечание: еще раз напомним, что вы можете раздавать эти файлы любым сервером. Главное – их доступность по заданному url-адресу. Просто за счет дополнительной константы `STATICFILES_DIRS` можно на период разработки сделать это силами сервера Django.

Часто путают смысл констант:

- `STATIC_URL` – это путь по сети (доступ из браузера);
- `STATICFILES_DIRS` – это пути к статическим файлам на жестком диске (доступ из файловой системы).

При развертывании на боевом сервере константу `STATICFILES_DIRS` можно удалить – она не имеет смысла.

Итак, теперь нам необходимо во всех шаблонах скорректировать ссылки на статические файлы.

Динамические url-ссылки в шаблонах

Теперь наведем порядок с гиперссылками в шаблонах. Для этого пропишем третий именованный аргумент `name` в функциях `url()` в файле `urls.py`:

```
urlpatterns = [
    url(r'^$', mainapp.main, name='main'),
    url(r'^products/', mainapp.products, name='products'),
    url(r'^contact/', mainapp.contact, name='contact'),
    url(r'^admin/', admin.site.urls),
]
```

Это значит, мы каждому адресу ставим в соответствие некоторое уникальное имя. Теперь в шаблонах можно записывать гиперссылки следующим образом:

```
<a href="{% url 'products' %}">п р о д у к т ы</a>
```

Раньше писали:

```
<a href="/products/">п р о д у к т ы</a>
```

Благодаря тегу `{% url '<имя_адреса_в_urls.py>' %}` в соответствии с регулярным выражением в `urls.py` при рендеринге шаблона адрес гиперссылки формируется динамически. Если мы изменим регулярку – реальный адрес на странице изменится автоматически.

Теперь нет необходимости делать большое количество правок в шаблонах при изменении схемы URL-адресов сайта — нужно **только** отредактировать `urls.py`.

Как видим, фреймворк избавляет нас от рутины при работе со статикой и гиперссылками.

Наследование шаблонов в Django

Если посмотреть на наши html-файлы страниц — можно заметить много общего кода: загрузка статики, почти одинаковое меню, footer. Любой избыточный код (на сленге: *copy-paste*) — это плохо. Любое изменение необходимо делать сразу в нескольких файлах, что снижает надежность и скорость разработки.

В современных фреймворках существует механизм, позволяющий решить и эту проблему: наследование шаблонов. Суть идеи в том, что на основе анализа кода страниц формируется некоторый базовый шаблон (например, `base.html`), включающий их общую часть, а в страницах остается только уникальная составляющая.

Наследование шаблонов позволяет создать основной «скелетный» шаблон, который содержит все общие части вашего сайта и определить «блоки», которые могут быть заменены шаблонами-наследниками. Для выделения структурных блоков в базовом шаблоне используется тег

```
{% block <имя блока> %}:
    {% block title %}
Главная
{% endblock %}
```

Если прописать этот тег в шаблоне-наследнике, то содержимое блока, которое было в базовом шаблоне, **заменится** новым. Если вы не прописываете какой-нибудь блок в шаблоне-наследнике — его содержимое будет как в базовом шаблоне. За счет такого подхода количество кода становится минимальным.

Замечание: если вы хотите **дополнить**, а не **заменить** содержимое блока из базового шаблона — пропишите в блоке шаблона-наследника: `{{ block.super }}` — это по сути переменная с контентом блока из родительского шаблона.

В нашем случае базовый шаблон может иметь примерно следующую структуру.

base.html

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
<head>
    <meta charset="utf-8">
    <title>
        {% block title %}
            {{ title|title }}
        {% endblock %}
    </title>
    {% block css %}
        <link rel="stylesheet" href="{% static 'css/style.css' %}">
        <link rel="stylesheet" href="{% static
'fonts/font-awesome/css/font-awesome.css' %}">
    {% endblock %}

    {% block js %}
    {% endblock %}
</head>
<body>
    <div class="container">
        {% block menu %}

        {% endblock %}

        {% block content %}

        {% endblock %}

    </div>

    {% block footer %}
        <div class="footer">
            <div class="text-footer">
                <h3>контакты</h3>
            </div>
            <div class="text-footer">
                <h3>полезная информация</h3>
            </div>
        </div>
    {% endblock %}

</body>
</html>
```

Замечание: некоторые блоки сделаны пустыми — это задел на будущее. Если в шаблоне-наследнике прописать блок, которого нет в базовом шаблоне, — он **не будет отображаться в браузере**. Если вы пишете что-либо вне блоков в шаблонах-наследниках — это тоже не отображается. Будьте внимательны!

В начале каждого шаблона-наследника необходимо прописать тег `{% extends ' <путь_к_базовому_шаблону>' %}`, например:

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

В большинстве случаев, аргументом для тега `{% extends %}` будет строка. Но может быть и переменная, если вы не знаете имя основного шаблона до запуска приложения. Это позволяет вам реализовывать динамические элементы.

Есть особенность шаблонизатора — теги не наследуются. Это значит, что тег `{% load staticfiles %}` нужно прописывать во всех шаблонах.

Подключение подшаблонов в Django

При разработке интернет-магазина мы можем столкнуться с ситуацией, когда на страницах есть элементы, имеющие и общее, и разное оформление или структуру (например, меню или весь header). Их не получится разместить в базовом шаблоне. Чтобы избавиться от избыточного кода можно общую часть вынести в отдельный файл (назовем его «подшаблон») и подключить при помощи тега `{% include ' <путь_к_подшаблону>' %}`. На практике мы поступим так с меню. Для порядка лучше файлы подшаблонов именовать особым образом и хранить в отдельной папке. Мы, например, будем делать префикс `inc_` и хранить в папке `'/includes/'`.

Передача контекста в шаблон

Мы подходим к самой важной части урока — динамический контент на страницах. Тут можно выделить две задачи:

- получение контента в контроллере;
- передача из контроллера в шаблон.

В контроллер контент должен попадать из моделей (базы данных). Но модели мы будем создавать позже на курсе. Поэтому в данном уроке рассмотрим два альтернативных временных варианта:

- прописать прямо в контроллере в виде списков или словарей;
- загрузить из файла (например, в формате «json»).

Реализацию этих подходов рассмотрим на практике.

Для передачи контекста (синоним слова контент) в шаблон необходимо в файле `views.py` прописать еще один позиционный аргумент (пусть это будет переменная `content`) в функции `render()`:

```
render(request, 'mainapp/index.html', content)
```

Переменная `content` должна быть словарем. Ключи этого словаря становятся в шаблоне именами переменных, а значения — значениями этих переменных, которые можно вывести на странице. Мы рассмотрим работу этого механизма в практической части урока на примере списка продуктов магазина, сведений об организации и меню.

Самая главная сложность на данном этапе — привыкнуть к доступу к элементам списка в шаблонах. Индекс пишем не в квадратных скобках, а через точку:

```
users_list[0] -> users_list.0
```

Точно так же перестраиваем мышление при работе со словарями.

Закрепление материала

Давайте еще раз посмотрим, как работает Django. Пусть, например, пользователь выбрал элемент меню «продукты» нашего магазина. Что произойдет?

1. Придет запрос на сервер. Диспетчер URL будет просматривать список `urlpatterns` в файле `urls.py` до первого совпадения запрашиваемого адреса (`products/`) с регулярным выражением. Это произойдет в строке:

```
url(r'^products/', mainapp.products, name='products'),
```

2. Поэтому будет вызван контроллер `products` из файла `mainapp.views.py` и ему в качестве аргумента передан объект запроса `request`.
3. В контроллере формируются (в будущем подгружаются из моделей) значения переменных `title` (заголовок страницы), `same_products` (похожие продукты) и `links_menu` (меню категорий).
4. Переменные помещаются в словарь контекста (напомним, что в нашем проекте это переменная `content`):

```
content = {
    'title': title,
    'links_menu': links_menu,
    'same_products': same_products
}
```

5. Работа контроллера завершается рендерингом шаблона `'mainapp/products.html'` и отправкой (`return`) ответа браузеру клиента:

```
return render(request, 'mainapp/products.html', content)
```

На следующих уроках мы добавим логику в контроллеры и будем создавать модели.

*Работа с меню

Обычно на страницах сайта выбранный пункт меню должен особым образом выделяться. Для этого, как правило, используют класс `active`. Можно по-разному реализовать автоматическое переключение этого класса между элементами DOM модели меню:

- при помощи передачи имени текущей страницы в контекст шаблона;
- при помощи JavaScript;
- использовать возможности Django.

Рассмотрим в качестве примера меню категорий продуктов на странице «продукты». Как уже было сказано выше, в шаблон под именем `links_menu` передается список:

```
links_menu = [
    {'href': 'products_all', 'name': 'все'},
    {'href': 'products_home', 'name': 'дом'},
    {'href': 'products_office', 'name': 'офис'},
    {'href': 'products_modern', 'name': 'модерн'},
    {'href': 'products_classic', 'name': 'классика'},
]
```

Подшаблон этого меню имеет следующий код:

`inc_categories_menu.html`

```
<ul class="links-menu">
  {% for link in links_menu %}
    <li>
      <a href="{% url link.href %}"
        class="{% if request.resolver_match.url_name == link.href %}
          active
        {% endif %}">
        {{ link.name }}
      </a>
    </li>
  {% endfor %}
</ul>
```

Этот код может показаться сложным — не обязательно сразу его использовать. Но вид страницы станет гораздо лучше, если меню работает правильно. Разберем по шагам.

Благодаря циклу, пробегаем все значения списка `links_menu`: как было показано выше — это словари с двумя ключами `'href'` и `'name'`.

При помощи ключа `'href'` формируем динамически гиперссылку (напоминаем: доступ к ключам словаря — через точку!):

```
href="{% url link.href %}"
```

Самое важное: мы можем получить url-адрес, по которому перешел пользователь, прямо в шаблоне — это значение `resolver_match.url_name` объекта `request`.

Дальше все просто: если значение совпало с адресом текущего элемента меню — ставим класс `active`.

Мы еще вернемся к атрибуту `resolver_match` на следующих уроках курса, когда доберемся до пространств имен в url-адресах.

Домашнее задание

Необходимо реализовать все изменения в своем проекте.

1. Реализовать наследование шаблонов в проекте. Меню подключить как подшаблон.

2. Реализовать в проекте механизмы работы со статическими файлами и url-адресами, которые прошли на уроке.
3. Поработать с шаблонными тегами и фильтрами (заглавные буквы, вывод текущей даты в шаблоне).
4. Организовать вывод динамического контента на страницах (элементы меню, список товара, заголовков страницы).
5. * Организовать загрузку динамического контента в контроллеры с жесткого диска (например, в формате «json»).

Все проблемы и пути их решения, подробно обсудим на следующем занятии.

Дополнительные материалы

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

1. [Шаблоны в Django](#)
2. [Статические файлы](#)
3. [Язык шаблонов и наследование](#)
4. [Шаблонные теги и фильтры](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Объект request \(english\)](#)