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

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

Веб-фреймворк Django позволяет динамически генерировать HTML. Самый распространенный подход — использование шаблонов. Они 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). Они дают гибкость в конфигурировании и позволяют реализовывать требуемую функциональность.

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

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

Задача бэкенда — загрузить шаблон и выполнить обработку с учетом контекста. В результате получается готовая html-страница, которую сервер и возвращает пользователю.

Замечание: при необходимости можно подключить другой бэкенд — например, **‘Jinja2’.** Мы не будем менять настройку и оставим **‘DjangoTemplates’**.

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

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

В Python для передачи **индекса** или имени **ключа** используем квадратные скобки:

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

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

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

Также язык шаблонизатора Django позволяет преобразовывать переменные (хотя по идее вся обработка должна происходить на языке Python в файлах **views.py**). Эти преобразования называются **шаблонными фильтрами**. Сначала это может показаться непривычным.

По сути шаблонный фильтр — это просто функция, написанная на Python, которая возвращает измененные данные в точку вызова. В конце курса 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 могут привести к ошибке.

Обязательно почитайте оригинальную документацию 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.

Теперь необходимо во всех шаблонах скорректировать ссылки на статические файлы.

Динамические url-ссылки в шаблонах

Наведем порядок с гиперссылками в шаблонах. Для этого пропишем третий именованный аргумент **name** в функциях **path()** в файле **urls.py**:

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

import mainapp.views as mainapp

urlpatterns = [
    path('', mainapp.main, name='main'),
    path('products/', mainapp.products, name='products'),
    path('contact/', mainapp.contact, name='contact'),
    path('admin/', admin.site.urls),
]
```

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

```
<a href="{% url 'products' %}">продукты</a>
```

Раньше писали:

```
<a href="/products/">продукты</a>
```

Благодаря тегу **{% url '<имя_адреса_в_urls.py>' %}** в соответствии с путями в **urls.py** при рендеринге шаблона адрес гиперссылки формируется динамически. Если изменим путь, реальный адрес на странице изменится автоматически.

Теперь не надо делать много правок в шаблонах при изменении схемы URL-адресов сайта — нужно только отредактировать **urls.py**.

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

Наследование шаблонов в Django

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

В современных фреймворках есть механизм, позволяющий решить и проблему наследования шаблонов. Идея в том, что на основе анализа кода страниц формируется 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 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 таким же, как в базовом. 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 такую 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 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 можем столкнуться с тем, что у ряда элементов есть и общее, и разное оформление или структура (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 поступим так с меню. Для порядка лучше файлы подшаблонов именовать особенным образом и хранить в отдельной папке. Будем делать префикс **inc_** и хранить в папке **'/includes/'**.

Передача контекста в шаблон

Мы подходим к самой важной части урока — динамическому контенту на страницах. 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. Его ключи становятся в шаблоне именами переменных, а значения — значениями этих переменных, которые можно вывести на странице. Рассмотрим работу этого механизма в практической части урока на примере списка продуктов магазина, сведений об организации и меню.

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
```

Также перестраиваем мышление при работе со словарями.

Закрепление материала

Еще раз посмотрим, как работает Django. Если пользователь выберет элемент меню «Продукты», что произойдет?

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

   ```
   path('products/', mainapp.products, name='products'),
   ```

2. Поэтому будет вызван контроллер **products** из файла **mainapp.views.py** и ему в качестве аргумента передастся 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 (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', context=content)
   ```

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

* Работа с меню

Обычно на страницах сайта выбранный пункт меню должен 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». В шаблон **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 страница выглядит лучше, если меню работает правильно. Разберем по шагам.

С помощью цикла пробегаем все значения списка **links_menu**: это словари с ключами **'href'** и **'name'**.

При помощи ключа **'href'** динамически формируем гиперссылку (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, когда доберемся до пространств имен в url-адресах.

Практическое 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, заголовок 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 чем 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).