Материалы занятия

Курс: Разработка программного обеспечения
Дисциплина: Создание web-приложений с использованием фреймворка Django

Тема занятия № 28: Шаблоны: Расширенные инструменты

1. НАПИСАНИЕ СВОИХ ФИЛЬТРОВ И ТЕГОВ

Здесь рассказывается, как написать свой собственный фильтр и самую простую разновидность тега (более сложные разновидности приходится разрабатывать много реже).

Организация исходного кода

Модули с кодом, объявляющим фильтры и теги шаблонизатора, должны находиться в пакете templatetags пакета приложения. Поэтому сразу же создадим в пакете приложения папку с именем templatetags, а в ней — ''пустой" модуль__init__.py.

Внимание!

Если отладочный веб-сервер Django запущен, после создания пакета templatetags его следует перезапустить, чтобы он перезагрузил обновленный код сайта с вновь созданными модулями.

Вот схема организации исходного кода для приложения bboard (предполагается, что код фильтров и тегов хранится в модуле filtersandtags.py):

```
<папка проекта>
    bboard
        __.init__.py
        . . .
        templatetags
            __.init__.py
            filtersandtags.py
        . . .
```

Каждый модуль, объявляющий фильтры и теги, становится библиотекой тегов. псевдоним этой библиотеки совпадает с именем модуля.

Написание фильтров

Проще всего написать фильтр, который принимает какое-либо значение и, возможно, набор параметров, после чего возвращает то же самое значение в преобразованном виде.

Написание и использование простейших фильтров

Фильтр — это обычная функция Django, которая:
□ в качестве первого параметра принимает обрабатываемое значение;
□ в качестве последующих параметров принимает значения параметров, указанных у фильтра. Эти параметры могут иметь значения по умолчанию;
□ возвращает в качестве результата преобразованное значение.
Объявленную функцию нужно зарегистрировать в шаблонизаторе в качестве фильтра.
Сначала необходимо создать экземпляр класса Library ИЗ МОДУЛЯ django. Template.
Потом у этого экземпляра класса нужно вызвать метод filter о в следующем формате:

```
filter(<имя регистрируемого фильтра>,
       <ссылка на функцию, реализующую фильтр>)
```

Зарегистрированный фильтр будет доступен в шаблоне под указанным именем.
Приведен пример объявления и регистрации фильтра currency. Он принимает числовое значение и, в качестве необязательного параметра, обозначение денежной единицы. В качестве результата он возвращает строку с числовым значением, отформатированным как денежная сумма.

```
from django import template

register = template.Library()

def currency(value, name='руб.'):
    return '%1.2f %s' % (value, name)

register.filter('currency', currency)
```

Вызов метода filter о можно оформить как декоратор. В таком случае он указывается у функции, реализующей фильтр, и вызывается без параметров. Пример:

```
@register.filter
def currency(value, name='руб.'):
    . . .
```

В необязательном параметре name декоратора filter о можно указать другое имя, под которым фильтр будет доступен в шаблоне:

```
@register.filter(name='cur')
def currency(value, name='руб.'):
    . . .
```

Может случиться так, что фильтр в качестве обрабатываемого должен принимать значение исключительно строкового типа, но ему было передано значение, тип которого отличается от строки (например, число). В этом случае при попытке обработать такое значение как строку (скажем, при вызове у него метода, который поддерживается только строковым типом) возникнет ошибка. Но мы можем указать Django предварительно преобразовать нестроковое значение в строку. Для этого достаточно задать для функции, реализующей фильтр, декоратор stringfilter из модуля django.template.defaultfilters. Пример:

```
from django.template.defaultfilters import stringfilter
. . .
@register.filter
@stringfilter
def somefilter(value):
    . . .
```

Если фильтр в качестве обрабатываемого значения принимает дату и время, то мы можем указать, чтобы это значение было автоматически преобразовано в местное время в текущей временной зоне. Для этого нужно задать у декоратора filter параметр expects localtime со значением True. Пример:

```
@register.filter(expects_localtime=True)
def datetimefilter(value):
    . . .
```

Объявленный фильтр можно использовать в шаблонах. Ранее говорилось, что модуль, объявляющий фильтры, становится библиотекой тегов, псевдоним которой совпадает с именем модуля. Следовательно, чтобы задействовать фильтр, нужно предварительно загрузить нужную библиотеку тегов с помощью тега load.
Пример (предполагается, что фильтр currency объявлен в модуле filtersandtags.py):

```
{% load filtersandtags %}
```

Объявленный нами фильтр используется так же, как и любой из встроенных в Django:

```
{{ bb.price|currency }}
{{ bb.price|currency:'p.' }}
```

Управление заменой недопустимых знаков HTML

Если в выводимом на страницу значении присутствует какой-либо из недопустимых знаков HTML: символ ''меньше", ''больше", двойная кавычка, амперсанд, то он должен быть преобразован в соответствующий ему специальный символ. В коде фильтров для этого можно использовать две функции из модуля django.utiis.html:

□ escape(<строка>) — выполняет замену всех недопустимых знаков в строке и возвращает обработанную строку в качестве результата;

□ conditional_escape (<строка>) — ТО же самое, ЧТО escape (), НО выполняет замену только в том случае, если в переданной ему строке такая замена еще не производилась.

Результат, возвращаемый фильтром, должен быть помечен как строка, в которой была выполнена замена недопустимых знаков. Сделать это можно, вызвав функцию Ma rk_safe (<помечаемая строка>} ИЗ модуля django. Utils. Safestring и вернув ИЗ фильтра возвращенный ей результат. Пример:

```
from django.utils.safestring import mark_safe
. . .
@register.filter
def somefilter(value):
    . . .
    return mark_safe(result_string)
```

Строка, помеченная как прошедшая замену недопустимых знаков, представляется экземпляром класса safetext ИЗ ТОГО же модуля django.utils.safestring. Так что мы можем проверить, проходила ли полученная фильтром строка процедуру замены или еще нет. Пример:

```
from django.utils.html import escape
from django.utils.safestring import SafeText
. . .
@register.filter
def somefilter(value):
    if not isinstance(value, SafeText):
        # Полученная строка не прошла замену. Выполняем ее сами
        value = escape(value)
    . . .
```

Еще нужно учитывать тот факт, что разработчик может отключить автоматическую замену недопустимых знаков в каком-либо фрагменте кода шаблона, заключив его в тег шаблонизатора autoescape . . . Endautoescape. Чтобы в коде фильтра выяснить, была ли отключена автоматическая замена, следует указать в вызове декоратора filter о параметр

needs_autoescape со значением True и добавить в список параметров функции, реализующей фильтр, параметр autoescape со значением по умолчанию True. В результате последний получит значение True, если автоматическая замена активна, и False, если она была отключена. Пример:

```
@register.filter(needs_autoescape=True)
def somefilter(value, autoescape=True):
```

```
  if autoescape:
      value = escape(value)
  . . .
```

И наконец, можно уведомить Django, что он сам должен выполнять замену в значении, возвращенном фильтром. Для этого достаточно в вызове декоратора filter о указать параметр is safe со значением True. Пример:

```
@register.filter(is_safe=True)
def currency(value, name='руб.'):
    . . .
```

Написание тегов

Объявить простейший одинарный тег шаблонизатора, вставляющий какие-либо данные в то место, где он находится, немногим сложнее, чем создать фильтр.

Написание тегов, выводящих элементарные значения

Если объявляемый тег должен выводить какое-либо элементарное значение: строку, число или дату, — нужно лишь объявить функцию, которая реализует этот тег.

Функция, реализующая тег, может принимать произвольное количество параметров, как обязательных, так и необязательных. В качестве результата она должна возвращать выводимое значение в виде строки.

Подобного рода тег регистрируется созданием экземпляра класса Library из модуля Template И ВЫЗОВОМ у ЭТОГО экземпляра метода simple_tag([name=None] [,] [takes_Context=Faise]), причем вызов нужно оформить в виде декоратора у функции, реализующей тег.

Пример объявления тега 1st. Он принимает произвольное количество параметров, из которых первый — строка-разделитель — является обязательным, выводит на экран значения остальных параметров, отделяя их друг от друга строкой-разделителем, а в конце ставит количество выведенных значений, взятое в скобки.

```
from django import template

register = template.Library()

@register.simple_tag
def lst(sep, *args):
    return '%s (итого %s)' % (sep.join(args), len(args))
```

По умолчанию созданный таким образом тег доступен в коде шаблона под своим изначальным именем, которое совпадает с именем функции, реализующей тег.
В вызове метода simpie tago мы можем указать два необязательных именованных параметра:
□ name— имя, под которым тег будет доступен в коде шаблона. Используется, если нужно указать для тега другое имя;
□ takes context— если True, то первым параметром в функцию, реализующую тег, будет передан контекст шаблона:

```
@register.simple_tag(takes_context=True)
def lst(context, sep, *args):
    . . .
```

Значение, возвращенное таким тегом, подвергается автоматической замене недопустимых знаков HTML на специальные символы. Так что нам самим это делать не придется.
Если же замену недопустимых знаков проводить не нужно (например, написанной нами тег должен выводить фрагмент HTML-кода), то возвращаемое функцией значение можно ’’пропустить” через функцию mark safe ().
Объявив тег, мы можем использовать его в шаблоне (не забыв загрузить модуль, в котором он реализован):

```
{% load filtersandtags %}
. . .
{% lst ', ' '1' '2' '3' '4' '5' '6' '7' '8' '9' %}
```

Написание шаблонных тегов
Если тег должен выводить фрагмент HTML-кода, то мы можем, как говорилось ранее, "пропустить" возвращаемую строку через функцию mark safe (). Вот пример подобного рода тега:

```
@register.simple_tag
def lst(sep, *args):
    return mark_safe('%s (итого <strong>%s</strong>)' %
                     (sep.join(args), len(args)))
```

Но если нужно выводить более сложные фрагменты HTML-кода, то удобнее объявить шаблонный тег. Возвращаемое им значение формируется так же, как и обычная веб-страница Django-сайта, — рендерингом на основе шаблона.
Функция, реализующая такой тег, должна возвращать в качестве результата контекст шаблона. В качестве декоратора, указываемого для этой функции, нужно поместить вызов метода inciusion_tag() экземпляра класса Library. Вот формат вызова этого метода:

```
inclusion_tag(<путь к шаблону>[, name=None][, takes_context=False])
```

Код объявляет шаблонный тег ulist. Он аналогичен объявленному ранее тегу 1st, но выводит перечень переданных ему позиций в виде маркированного списка HTML, а количество позиций помещает под списком и выделяет курсивом.

```
from django import template

register = template.Library()

@register.inclusion_tag('tags/ulist.html')
def ulist(*args):
    return {'items': args}
```

Шаблон такого тега ничем не отличается от шаблонов веб-страниц и располагается непосредственно в папке templates пакета приложения. Код шаблона tags\ulist.html
Тега ulist приведен.

```
<ul>
    {% for item in items %}
    <li>{{ item }}</li>
    {% endfor %}
</ul>
<p>Итого <em>{{ items|length }}</em></p>
```

Метод inclusion tag o поддерживает необязательные параметры name и takes_Context. При указании значения True для параметра Takes context контекст шаблона страницы также будет доступен в шаблоне тега.
Используется шаблонный тег так же, как и обычный, возвращающий элементарное значение:

```
{% load filtersandtags %}
. . .
{% ulist '1' '2' '3' '4' '5' '6' '7' '8' '9' %}
```

На заметку!
Django также поддерживает объявление более сложных тегов, являющихся парными и выполняющих над своим содержимым различные манипуляции (в качестве примера можно привести тег for . . . Endfor).
Интересующиеся могут найти руководство по этой теме на странице:
Https://docs.djangoproject.eom/en/3.0/howto/custom-template-tags/.

Регистрация фильтров и тегов

Если мы, согласно принятым в Django соглашениям, сохранили модуль с объявлениями фильтров и тегов в пакете templatetags пакета приложения, ничего более делать не нужно. Фреймворк превратит этот модуль в библиотеку тегов, имя модуля станет псевдонимом этой библиотеки, и нам останется лишь загрузить ее, воспользовавшись тегом load шаблонизатора. Но если мы не последовали этим соглашениям или хотим указать у библиотеки тегов другой псевдоним, то придется внести исправления в настройки проекта, касающиеся обработки шаблонов.
Чтобы зарегистрировать модуль с фильтрами и тегами как загружаемую библиотеку тегов (т. Е. Требующую загрузки тегом load шаблонизатора), ее нужно добавить в список загружаемых библиотек. Этот список хранится в дополнительных настройках шаблонизатора, задаваемых параметром options в параметре libraries.
Предположим, что модуль filtersandtags.py, хранящий фильтры и теги, находится непосредственно в пакете приложения (что нарушает соглашения Django). Тогда зарегистрировать его мы можем, записав в модуле settings.py пакета конфигурации такой код (выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            . . .
            'libraries': {
                'filtersandtags': 'bboard.filtersandtags',
            }
        },
    },
]
```

Нам даже не придется переделывать код шаблонов, т. К. Написанная нами библиотека тегов будет Доступна ПОД Тем Же псевдонимом filtersandtags.
Мы можем изменить псевдоним этой библиотеки тегов, скажем, на ft:

```
'libraries': {
    'ft': 'bboard.filtersandtags',
}
```

И сможем использовать для ее загрузки новое, более короткое имя:

```
{% load ft %}
```

Если же у нас нет желания писать в каждом шаблоне тег load, чтобы загрузить библиотеку тегов, то мы можем оформить ее как встраиваемую— и объявленные в ней фильтры и теги станут доступными без каких бы то ни было дополнительных действий. Для этого достаточно указать путь к модулю библиотеки тегов в списке дополнительного параметра builtins. Вот пример (добавленный код выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            . . .
```

```
            'builtins': [
                'bboard.filtersandtags',
            ],
        },
    },
]
```

После этого мы сможем просто использовать все объявленные в библиотеке теги, когда и где нам заблагорассудится.

2. ПЕРЕОПРЕДЕЛЕНИЕ ШАБЛОНОВ

Предположим, нужно реализовать выход с сайта с выводом страницы с сообщением о выходе. Для этого мы решаем использовать стандартный контроллер-класс Logoutview и указанный для него по умолчанию шаблон Registration\logged_out.html. Мы пишем шаблон logged_out.html, помещаем его в папку Registration, вложенную в папку templates пакета приложения, запускаем отладочный веб-сервер, выполняем вход на сайт, переходим на страницу выхода. И наблюдаем на экране не нашу страницу, а какую-то другую, судя по внешнему виду, принадлежащую административному сайту Django.

Дело в том, что Django в поисках нужного шаблона просматривает папки templates, находящиеся в пакетах всех приложений, которые зарегистрированы в проекте, и прекращает поиски, как только найдет первый подходящий шаблон. В нашем случае таким оказался шаблон registration\logged_out.html, принадлежащий стандартному приложению django. Contrib.admin, т. Е. Административному сайту.

Мы можем избежать этой проблемы, просто задав для контроллера-класса шаблон с другим именем. Это можно сделать либо в маршруте, вставив нужный параметр в вызов метода as viewo контроллера-класса, либо в его подклассе, в соответствующем атрибуте. Но существует и другой способ — использовать переопределение шаблонов, ’’подсунув” Django наш шаблон до того, как он доберется до стандартного.

Есть два способа реализовать переопределение шаблонов:

□ приложения в поисках шаблонов просматриваются в том порядке, в котором они указаны в списке зарегистрированных приложений из параметра installed apps настроек проекта. Следовательно, мы можем просто поместить наше приложение перед стандартным.

Пример (предполагается, что нужный шаблон находится в приложении bboard):

```
INSTALLED_APPS = [
    'bboard',
    'django.contrib.admin',
    . . .
]
```

После этого, поместив шаблон registration\logged_out.html в папку templates пакета приложения bboard, мы можем быть уверены, что наш шаблон будет найден раньше, чем стандартный;

□ папки, пути к которым приведены в списке параметра dirs настроек шаблонизатора, просматриваются перед папками templates пакетов приложений. Мы можем создать в папке проекта папку main_templates и поместить шаблон registration\logged_out.html в нее. После чего нам останется изменить настройки шаблонизатора следующим образом:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'main_templates')],
        . . .
    },
]
```

Значение переменной base dir вычисляется в модуле settings.py ранее и представляет собой полный путь к папке проекта.

Здесь мы указали в списке параметра dirs единственный элемент— путь к только что созданной нами папке. И опять же, мы можем быть уверены, что контроллер-класс будет использовать наш, а не ’’чужой” шаблон.

Однако при переопределении шаблонов нужно иметь в виду один весьма неприятный момент. Если мы переопределим какой-либо шаблон, задействуемый стандартным приложением, то это стандартное приложение будет использовать переопределенный нами шаблон, а не свой собственный. Например, если мы переопределим шаблон registration\logged_out.html, принадлежащий стандартному приложению Django.contrib.admin, то последнее будет использовать именно переопределенный шаблон. Так что в некоторых случаях, возможно, будет целесообразнее воздержаться от переопределения шаблонов стандартных приложений и написать свой шаблон.