# Об этом сайте

Этот сайт в основном посвящён программированию и создан для программистов, в первую очередь начинающих. Содержание сайта основано на разборе задач автоматизации, с которыми мне приходится сталкиваться в жизни, и которые по моему мнению представляют интерес для аудитории более широкой, нежели я сам. Здесь я расскажу о своих подходах к решению указанных задач. Естественно, тривиальные методы, ясно и полно освещённые в стандартной документации и в других источниках, дублироваться в полном объёме не будут. Конструктивный отклик на мои публикации приветствуется категорически.

Рассматриваемые инструментальные средства решения задач – это прежде всего Python и созданные на его базе открытые фреймворки (в моём случае это веб-фреймворк Django, который я использую чаще всего в качестве альтернативы Oracle APEX’u) и библиотеки, а также некоторые полезные проприетарные продукты, без которых порой обойтись сложно (например, хорошо известные продукты Oracle).

Содержимое сайта не будет ограничено указанными темами, я планирую написать что-нибудь интересное и по другим темам, представляющим для меня интерес – информационная безопасность, управление качеством ПО, практика применения открытого ПО в крупных промышленных проектах и прочим.

# Работа с шаблонами Django: использование RequestContext и Context без использования файлов

Типичная техника отображения, проиллюстрированная в документации, выглядит примерно так:

from django.template.loader import get\_template

from django.template import Context

from django.http import HttpResponse

import datetime

def view1(request):

t = get\_template('my\_template.html')

topic = ‘О вреде курения’

html = t.render(Context({'topic':topic}))

return HttpResponse(html)

То же самое можно записать в одну строку, если использовать обёртку render\_to\_response() (понадобится дополнительно проимпортировать её из пакета django.shortcuts ):

…

from django.shortcuts import render\_to\_response

def view1(request):

topic = ‘О вреде курения’

return render\_to\_response(‘my\_template.html’, {‘topic’:topic})

В первом и втором примерах мы читаем шаблон из файла файловой системы. Такой подход безусловно можно рекомендовать в большинстве случаев, когда есть возможность достаточно легко откорректировать либо заменить файл, содержащий шаблон, который обычно расположен в файловой системе веб-сервера. Однако что делать, если наш сайт расположен на сервере виртуального хостинга и доступ к нему возможен только по протоколу SSH, а Вы большую часть времени проводите на работе в офисной сети, где доступ на внешние сайты по протоколу SSH закрыт? Варианта два – ждать окончания рабочего дня или обеспечить доступ к шаблонам по общедоступному протоколу, например http. Далее в этой статье мы остановимся на втором варианте.

C точки зрения Django класс Template – всего лишь обёртка для обычной строки, содержащей в себе управляющие команды и символы, которые показывают обработчику шаблонов, какие данные контекста и каким образом вставить в шаблон. Поэтому наш пример можно переписать и так:

**…**

from django.template import Context, Template

def view1(request):

topic = ‘О вреде курения’

t = Template(‘Много статей на тему “{{ topic }}”’)

c = Context({‘topic’:topic})

return HttpResponse(t.render(c))

У данного решения есть несколько недостатков:

1. Очевидно, данное решение не предполагает использование в шаблоне тэгов {% extends %} и {% include%};
2. Зарезервированные символы языка html, содержащиеся в переменной контекста, Django при обработке шаблона по умолчанию заменит на escape-последовательности);
3. Если код шаблона или переменная контекста содержат веб-форму, то мы получим ошибку обработки шаблона по причине отсутствия метки защиты от CSRF (Сross Site Request Forgery — подделка межсайтовых запросов).

А сейчас рассмотрим, насколько серьёзны перечисленные недостатки.

Первый недостаток легко устраняется последовательной обработку шаблонов, например:

…

def view2(request):

parent\_template = Template('<h1>{{ title }}</h1>{{ parent\_content }}')

child\_template = Template('Сегодня поговорим на тему "{{ child\_content }}".')

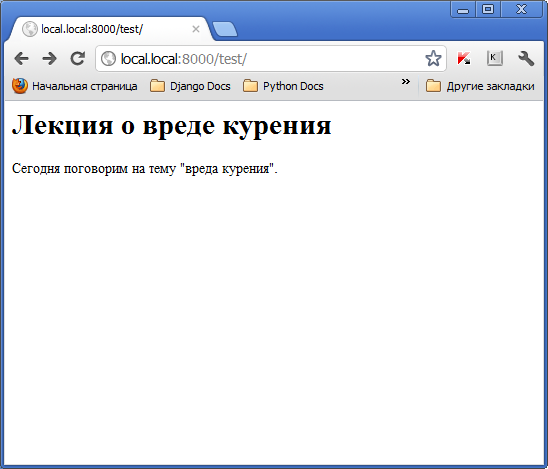
child\_context = Context({'child\_content':'вреда курения'})

parent\_context = Context({'parent\_content':child\_template.render(child\_context), 'title':'Лекция о вреде курения'})

return HttpResponse(parent\_template.render(parent\_context))

В приведённом коде для ясности использован нестрогий html-синтаксис.

Веб-сервер вернёт нам следующее:

****

Теперь немного изменим наш пример, выделив слова про вред курения жирным написанием, используя теги «<B>» и «</b>»:

def view2(request):

parent\_template = Template('<h1>{{ title }}</h1>{{ parent\_content }}')

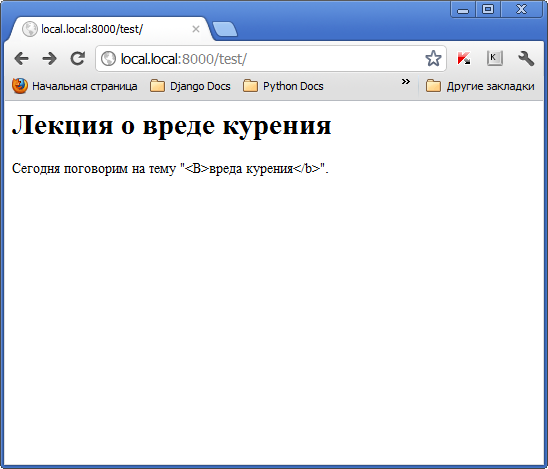
child\_template = Template('Сегодня поговорим на тему "{{ child\_content }}".')

child\_context = Context({'child\_content':'<B>вреда курения</b>'})

parent\_context = Context({'parent\_content':child\_template.render(child\_context), 'title':'Лекция о вреде курения'})

return HttpResponse(parent\_template.render(parent\_context))

В результате получим не совсем то, чего добивались:

****

Дело в том, что Django при обработке шаблонов заменяет зарезервированные в html символы escape-последовательностями, т.е. вместо «<B>» «</b>» подставляет «"&lt;B&gt;» и «&lt;/b&gt;».

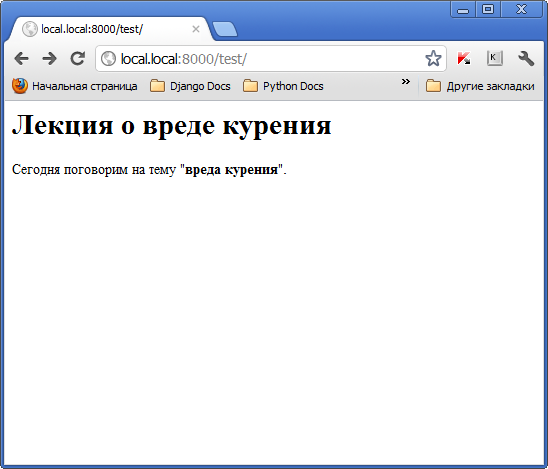
В нашем случае это необходимо отключить, заменив

child\_template = Template('Сегодня поговорим на тему "{{ child\_content }}".')

на

child\_template = Template('Сегодня поговорим на тему "{% autoescape off %}{{ child\_content }}{% endautoescape %}".')

после чего всё встаёт на свои места:



Теперь добавим в наш шаблон html-форму. Как известно из документации Django, шаблон в этом случае должен содержать после открывающего тэга <FORM> специальный блок, генерирующий скрытое поле формы с меткой защиты от CSRF: {% csrf\_token %}. Однако, следуя документации, для этого следует использовать для создания контекста экземпляр контекста, создаваемый при помощи класса RequestContext, причём пару ключ-значение для использования в метке защиты от CSRF необходимо получать явно при помощи функции csrf:

from django.core.context\_processors import csrf

…

context = {}

context.update(csrf(request))

context\_instance = RequestContext(request, context)

…

return HttpResponse(Template(<*строка шаблона с формой*>).render(t.Context(context\_instance))

Поэтому наше представление надо переписать следующим образом:

def view2(request):

if request.method=='POST':

# здесь производим обработку post-запроса

return redirect('/') # перенаправление для избежания повторной отправки post-запросов при обновлении страницы,корневой каталог взят для примера

# формируем экземпляр контекста

context = {}

context.update(csrf(request))

context\_instance = RequestContext(request, context)

#

parent\_template = Template('<h1>{{ title }}</h1>{{ parent\_content }}')

child\_template = Template('Сегодня поговорим на тему "{% autoescape off %}{{ child\_content }}{% endautoescape %}".' + \

'<FORM action="." method="POST">{% csrf\_token %}<TABLE>{{comment\_form.as\_table}}</table>' + \

'<input type="submit" value="Отправить"></form>')

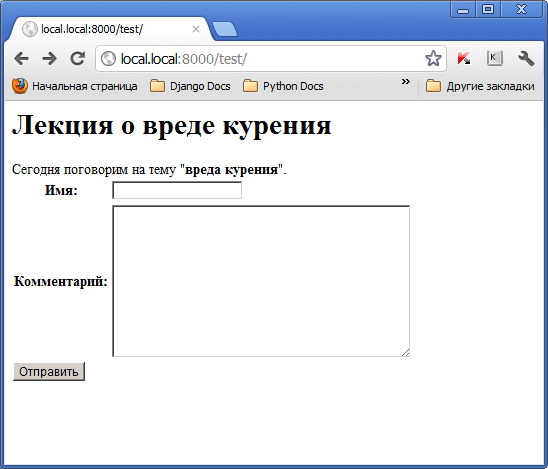
comment\_form=CommentForm()

context\_instance.update({'child\_content':'<B>вреда курения</b>', 'comment\_form':comment\_form})

parent\_context = Context({'parent\_content':child\_template.render(context\_instance), 'title':'Лекция о вреде курения'})

return HttpResponse(parent\_template.render(parent\_context))

В результате получаем то, что нам нужно, без использования шаблонов, сохранённых в файлах:



Следующий шаг – разместить нужные нам шаблоны в моделях Django и внутри наших представлений обращаться за шаблоном к ним. Возможно, это будет работать несколько медленнее канонической техники с файлами шаблонов, но зато мы можем практически без трудозатрат получить доступ к содержимому шаблонов, когда нам необходимо оперативно их откорректировать, например, если мы создаём систему управления контентом.