8

Углубленное изучение представлений и конфигурации URL

В главе 3 мы рассказали об основах работы с функциями представлений и конфигурацией URL в Django. В этой главе мы более подробно рассмотрим дополнительные возможности этих частей фреймворка.

Конфигурация URL: полезные приемы

В конфигурациях URL нет ничего особенного – как и все в Django, это просто программный код на языке Python. Это обстоятельство можно использовать разными способами.

Упрощение импорта функций

Рассмотрим следующую конфигурацию URL, созданную для примера из главы 3:

```
from django.conf.urls.defaults import *
from mysite.views import hello, current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^hello/$', hello),
    (r'^time/$', current_datetime),
    (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

Как объяснялось в главе 3, каждый элемент конфигурации URL включает функцию представления, которая передается в виде объекта-функции. Поэтому в начале модуля необходимо импортировать эти функции представления.

Но с увеличением сложности приложений Django растет и объем конфигурации URL, поэтому управлять инструкциями импорта становится утомительно. (Добавляя новое представление, нужно не забыть импор-

тировать его, и при таком подходе инструкция `import` очень скоро станет чрезмерно длинной.) Этого можно избежать, если импортировать сам модуль `views`. Следующая конфигурация URL эквивалентна предыдущей:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^hello/$', views.hello),
    (r'^time/$', views.current_datetime),
    (r'^time/plus/(d{1,2})/$', views.hours_ahead),
)
```

В Django существует еще один способ ассоциировать функцию представления с образцом URL: передать строку, содержащую имя модуля и функции вместо самого объекта-функции. Продолжим рассмотрение примера:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'mysite.views.hours_ahead'),
)
```

(Обратите внимание на кавычки, окружающие имена представлений. Мы написали `'mysite.views.current_datetime'` в кавычках, а не просто `mysite.views.current_datetime`.)

Этот прием позволяет избавиться от необходимости импортировать функции представлений; обнаружив строку, описывающую имя и путь к функции, Django автоматически импортирует функцию при первом упоминании.

При использовании описанного способа можно еще больше сократить программный код, вынеся вовне общий «префикс представления». В нашем примере конфигурации URL все строки начинаются с `'mysite.views'`, и записывать этот префикс каждый раз утомительно. Вместо этого можно передать его первым аргументом функции `patterns()`:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'hours_ahead'),
)
```

Обратите внимание на отсутствие завершающей точки (".") в префиксе и начальной точки в строке представления. Django подставит ее автоматически.

Какому из двух подходов отдать предпочтение, зависит от вашего стиля программирования и потребностей.

Запись полного пути в виде строки обладает следующими достоинствами:

- Она компактнее, так как не требует импортировать функции представления.
- Получающаяся конфигурация URL удобнее для чтения и сопровождения, если представления находятся в разных модулях Python.

Преимущества подхода на основе объектов-функций таковы:

- Он позволяет легко «обернуть» функции представления. См. раздел «Обертывание функций представления» ниже в этой главе.
- Он ближе к духу Python, точнее, к принятой в нем традиции передавать функции в виде объектов.

Оба подхода допустимы, их можно смешивать в одной и той же конфигурации URL. Выбор за вами.

Использование нескольких префиксов представлений

Применяя на практике прием передачи путей к представлениям в виде строк, вы вполне можете столкнуться с ситуацией, когда у представлений в конфигурации URL нет общего префикса. Но и в этом случае можно вынести префикс для устранения дублирования. Достаточно сложить несколько объектов `patterns()`.

Старый вариант:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(\d{1,2})/$', 'mysite.views.hours_ahead'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

Новый вариант:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
    (r'^time/plus/(\d{1,2})/$', 'hours_ahead'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

Для фреймворка достаточно, чтобы существовала переменная urlpatterns на уровне модуля. Ее можно конструировать и динамически, как показано в этом примере. Специально подчеркнем, что объекты, которые возвращает функция patterns(), можно складывать, хотя, возможно, это стало для вас неожиданностью.

Отладочная конфигурация URL

Узнав о возможности динамически конструировать переменную urlpatterns, вы, возможно, захотите воспользоваться этим и дополнить конфигурацию URL при работе с Django в режиме отладки. Для этого просто проверьте значение параметра DEBUG во время выполнения:

```
from django.conf import settings
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^$', views.homepage),
    (r'^(\d{4})/([a-z]{3})/$', views.archive_month),
)

if settings.DEBUG:
    urlpatterns += patterns('',
    (r'^debuginfo/$', views.debug),
)
```

Здесь URL /debuginfo/ доступен, только когда параметр DEBUG имеет значение True.

Именованные группы

До сих пор в регулярных выражениях всех примеров конфигураций URL встречались только простые, *неименованные* группы, то есть мы заключали интересующие нас части URL в скобки, а фреймворк Django передавал сохраняемый текст в функцию представления в виде позиционного параметра. Но существует также возможность использовать в регулярных выражениях *именованные* группы, позволяющие передавать сохраняемые части URL в виде *именованных* параметров.

Именованные и позиционные аргументы

В языке Python функциям можно передавать как именованные, так и позиционные аргументы, а в некоторых случаях те и другие одновременно. В случае вызова с именованным аргументом указывается не только передаваемое значение аргумента, но и его имя. При вызове с позиционным аргументом передается только значение – семантика аргументов определяется неявно, исходя из порядка их следования.

Рассмотрим такую простую функцию:

```
def sell(item, price, quantity):
   print "Продано %s единиц %s по цене %s" % (quantity, item, price)
```

При вызове функции с позиционными аргументами аргументы должны быть перечислены в том же порядке, в котором они описаны в определении функции:

```
sell('Носки', '$2.50', 6)
```

При вызове с именованными аргументами следует указать не только значения, но и имена аргументов. Все следующие инструкции эквивалентны:

```
sell(item='Носки', price='$2.50', quantity=6)
sell(item='Носки', quantity=6, price='$2.50')
sell(price='$2.50', item='Носки', quantity=6)
sell(price='$2.50', quantity=6, item='Носки')
sell(quantity=6, item='Носки', price='$2.50')
sell(quantity=6, price='$2.50', item='Носки')
```

Наконец, допускается передавать и позиционные, и именованные аргументы одновременно, при условии что все позиционные аргументы предшествуют именованным, например:

```
sell('Носки', '$2.50', quantity=6)
sell('Носки', price='$2.50', quantity=6)
sell('Носки', quantity=6, price='$2.50')
```

В регулярных выражениях для обозначения именованных групп применяется синтаксис (?P<name>pattern), где name – имя группы, а pattern – сопоставляемый образец. Ниже приведен пример конфигурации URL с неименованными группами:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', views.year_archive),
    (r'^articles/(\d{4})/(\d{2})/$', views.month_archive),
)
```

А вот та же конфигурация с именованными группами:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(?P<year>\d{4})/$', views.year_archive),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', views.month_archive),
)
```

В обоих случаях решается одна и та же задача, но с одним тонким отличием: сохраняемые значения передаются функциям представлений в виде именованных, а не позиционных аргументов.

Например, если используются неименованные группы, то обращение к URL /articles/2006/03/ приведет к такому вызову функции:

```
month_archive(request, '2006', '03')
```

В случае использования именованных групп функция будет вызвана так:

```
month_archive(request, year='2006', month='03')
```

С практической точки зрения использование именованных групп делает конфигурацию URL чуть более явной и менее подверженной ошибкам, связанным с неверным порядком следования аргументов. К тому же этот прием оставляет возможность переупорядочивать аргументы в определениях функций представлений. Так, если бы в предыдущем примере мы решили изменить структуру URL, поместив месяц *перед* годом, то при использовании неименованных групп пришлось бы поменять порядок аргументов в представлении `month_archive`. А воспользуйся мы именованными группами, порядок следования сохраняемых параметров в URL никак не отразился бы на представлении.

Разумеется, за удобство именованных групп приходится расплачиваться утратой краткости; некоторым разработчикам синтаксис именованных групп кажется уродливым и слишком многословным. Зато он проще для восприятия, особенно если код читает человек, плохо знакомый с использованием регулярных выражений в вашем приложении Django. Разобраться в конфигурации URL, в которой применяются именованные группы, можно с первого взгляда.

Алгоритм сопоставления и группировки

Недостаток именованных групп в конфигурации URL состоит в том, что в одном образце URL не могут одновременно встречаться именованные и неименованные группы. Если вы забудете это правило, то Django не сообщит ни о каких ошибках, но сопоставление URL с образцом будет происходить не так, как вы ожидаете. Поэтому опишем точный алгоритм анализа конфигурации URL в части обработки именованных и неименованных групп в регулярном выражении:

- Если имеются именованные аргументы, то используются только они, а неименованные игнорируются.
- В противном случае все неименованные аргументы передаются в виде позиционных параметров.
- В обоих случаях дополнительные параметры передаются в виде именованных аргументов. В следующем разделе приведена более подробная информация.

Передача дополнительных параметров функции представления

Иногда вы можете заметить, что написали две очень похожие функции представлений, отличающиеся лишь в мелких деталях. Например, единственным отличием может быть имя вызываемого шаблона:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

Код повторяется, а это некрасиво. Поначалу в голову приходит мысль избавиться от дублирования следующим образом: указать для обоих URL одно и то же представление, поместить URL в скобки, чтобы сохранить его целиком, а внутри функции проверить URL и вызвать подходящий шаблон:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^(foo)/$', views.foobar_view),
    (r'^(bar)/$', views.foobar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
```

```
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

Однако при таком решении возникает тесная связь между конфигурацией URL и реализацией представления. Если впоследствии вы решите переименовать /foo/ в /fooey/, то придется внести изменения в код представления.

Элегантный подход состоит в том, чтобы завести в конфигурации URL дополнительный параметр. Каждый образец URL может включать еще один (третий) элемент: словарь именованных аргументов, который передается функции представления.

С учетом этой возможности мы можем переписать код следующим образом:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foobar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foobar_view, {'template_name': 'template2.html'}),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

Как видите, в конфигурации URL определен дополнительный параметр `template_name`. Функция представления воспримет его как еще один аргумент.

Описанный прием – это элегантный способ передать дополнительную информацию функции представления с минимумом хлопот. Он используется в нескольких приложениях, поставляемых вместе с Django, и, прежде всего, в системе обобщенных представлений, которую мы рассмотрим в главе 11.

В следующих разделах предлагаются некоторые идеи, касающиеся использования дополнительных параметров в собственных проектах.

Имитация сохраняемых значений в конфигурации URL

Предположим, что имеется несколько представлений, которые отвечают некоторому образцу, и еще один URL, который не отвечает этому образцу, но логика представления для него такая же. В таком случае

можно «имитировать» сохранение частей URL путем использования дополнительных параметров в конфигурации URL, тогда этот выпадающий из общего ряда URL можно будет обработать в том же представлении, что и остальные.

Пусть, например, приложение отображает какие-то данные для каждого дня, и в нем используются URL такого вида:

```
/mydata/jan/01/
/mydata/jan/02/
/mydata/jan/03/
# ...
/mydata/dec/30/
/mydata/dec/31/
```

Пока все просто – нужно лишь сохранить переменные части в образце URL (применив именованные группы):

```
urlpatterns = patterns('',
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

А сигнатура функции представления будет такой:

```
def my_view(request, month, day):
    # ....
```

Пока ничего нового. Проблема возникает, когда нужно добавить еще один URL, для обработки которого желательно использовать то же представление `my_view`, однако в этом URL отсутствует параметр `month` или `day` (или оба сразу).

Допустим, что нам потребовалось добавить URL `/mydata/birthday/`, который был бы эквивалентен `/mydata/jan/06/`. На помощь приходит прием передачи дополнительных параметров:

```
urlpatterns = patterns('',
    (r'^mydata/birthday/$', views.my_view, {'month': 'jan', 'day': '06'}),
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

Прелесть в том, что при таком подходе вообще не пришлось изменять функцию представления. Она ожидает получить аргументы `month` и `day`, а откуда они возьмутся – из самого URL или из дополнительных параметров – ей безразлично.

Переход к обобщенным представлениям

В программировании считается хорошим тоном вычленять общий код. Например, имея такие две функции:

```
def say_hello(person_name):
    print 'Привет, %s' % person_name
```

```
def say_goodbye(person_name):
    print 'Пока, %s' % person_name
```

мы можем вычленить текст приветствия и сделать его параметром:

```
def greet(person_name, greeting):
    print '%s, %s' % (greeting, person_name)
```

Тот же прием можно применить к представлениям в Django, если воспользоваться дополнительными параметрами в конфигурации URL.

И тогда можно будет перейти к высокоуровневым абстракциям представлений. Не надо мыслить в терминах: «Это представление отображает список объектов `Event`, а то – список объектов `BlogEntry`». Считайте, что оба – частные случаи «представления, которое отображает список объектов, причем тип объекта – переменная».

Рассмотрим, к примеру, такой код:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^events/$', views.event_list),
    (r'^blog/entries/$', views.entry_list),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import Event, BlogEntry

def event_list(request):
    obj_list = Event.objects.all()
    return render_to_response('mysite/event_list.html',
        {'event_list': obj_list})

def entry_list(request):
    obj_list = BlogEntry.objects.all()
    return render_to_response('mysite/blogentry_list.html',
        {'entry_list': obj_list})
```

Оба представления делают по существу одно и то же: выводят список объектов. Так давайте вычленим тип отображаемого объекта:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import models, views

urlpatterns = patterns('',
    (r'^events/$', views.object_list, {'model': models.Event}),
    (r'^blog/entries/$', views.object_list, {'model': models.BlogEntry}),
)
```

```
# views.py

from django.shortcuts import render_to_response

def object_list(request, model):
    obj_list = model.objects.all()
    template_name = 'mysite/%s_list.html' % model.__name__.lower()
    return render_to_response(template_name, {'object_list': obj_list})
```

В результате небольшого изменения мы неожиданно получили повторно используемое, не зависящее от модели представление! Теперь всякий раз, как нам понадобится вывести список объектов, мы сможем воспользоваться представлением `object_list` и не писать новый код. Поясним, что же мы сделали.

- Мы передаем класс модели напрямую в виде параметра `model`. В словаре дополнительных параметров можно передавать объект Python любого типа, а не только строки.
- Строка `model.objects.all()`– это пример *динамической типизации* (duck typing – буквально, утиная типизация): «Если нечто переваливается, как утка, и крякает, как утка, то и обращаться с этим можно, как с уткой». Отметим, что функция ничего не знает о типе объекта `model`, ей достаточно, чтобы у него был атрибут `objects`, который в свою очередь должен иметь метод `all()`.
- При определении имени шаблона мы воспользовались методом `model.__name__.lower()`. Каждый класс в языке Python имеет атрибут `__name__`, который возвращает имя класса. Эту особенность удобно использовать в случаях, подобных нашему, когда тип класса не известен до момента выполнения. Например, для класса `BlogEntry` атрибут `__name__` содержит строку `'BlogEntry'`.
- Между этим примером и предыдущим есть небольшое отличие: мы передаем в шаблон обобщенное имя переменной `object_list`. Можно было бы назвать ее `blogentry_list` или `event_list`, но мы оставили это в качестве упражнения для читателя.

Поскольку у всех сайтов, управляемых данными, есть ряд общих характерных черт, в состав Django входит набор обобщенных представлений, в которых для экономии времени применяется описанная выше техника. Эти встроенные обобщенные представления мы рассмотрим в главе 11.

Конфигурационные параметры представления

Если вы распространяете свое приложение Django, то, скорее всего, пользователи захотят его настраивать. Предвидя это, имеет смысл предусмотреть в представлениях точки подключения, которые позволят изменять некоторые аспекты их поведения. Для этой цели можно воспользоваться дополнительными параметрами в конфигурации URL.

Очень часто высокая гибкость приложения достигается за счет настраиваемого имени шаблона:

```
def my_view(request, template_name):
   var = do_something()
   return render_to_response(template_name, {'var': var})
```

Приоритет дополнительных параметров над сохраняемыми значениями

В случае конфликта дополнительные параметры, указанные в конфигурации URL, имеют приоритет над извлеченными из URL при сопоставлении с регулярным выражением. Иными словами, если в образце URL имеется именованная группа и присутствует дополнительный параметр с таким же именем, то будет использован дополнительный параметр.

Рассмотрим, к примеру, такую конфигурацию URL:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id': 3}),
)
```

Здесь и в регулярном выражении, и в словаре дополнительных параметров имеется параметр с именем `id`. Тот `id`, что «зашит» в код, имеет приоритет. Это означает, что любой запрос к URL такого вида (например, `/mydata/2/` или `/mydata/432432/`) будет обрабатываться так, будто `id` равен 3 вне зависимости от того, какое значение извлечено из самого URL.

Проницательный читатель заметит, что в этом случае указывать `id` в именованной группе регулярного выражения – пустая трата времени, поскольку сохраняемое значение все равно будет затерто тем, что указано в словаре. Так оно и есть; мы привлекли внимание к этому обстоятельству только для того, чтобы помочь вам избежать этой ошибки.

Аргументы представления, принимаемые по умолчанию

Еще один удобный прием – определение значений по умолчанию для аргументов представления. Тем самым мы сообщаем представлению, какое значение параметра следует использовать, если оно явно не задано при вызове функции. Например:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
```

```
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num='1'):
    # Выводит страницу записей в блоге с номером num.
    # ...
```

Здесь оба образца URL указывают на одно и то же представление – `views.page`, – но в первом никакие части URL не сохраняются. Если будет обнаружено совпадение с первым образцом, то при вызове функции для аргумента `num` будет использовано значение по умолчанию – `'1'`. Если же со вторым, то функции будет передано сохраненное значение `num`.

Примечание

Мы специально определили в качестве значения аргумента по умолчанию строку `'1'`, а не целое число `1`, потому что сохраняемое значение всегда представлено строкой.

Как отмечалось выше, такой прием часто применяется в сочетании с конфигурационными параметрами. В следующем примере мы немного улучшим код примера из раздела «Конфигурационные параметры представления», определив значение по умолчанию для аргумента `template_name`:

```
def my_view(request, template_name='mysite/my_view.html'):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

Представления для обработки особых случаев

Иногда в конфигурации URL определяется образец, совпадающий сразу со многими URL, часть из которых нужно обрабатывать особым образом. Тогда можно воспользоваться тем фактом, что образцы просматриваются последовательно, и поместить особый случай в начало списка.

Например, страницы «добавить объект» в административном интерфейсе Django можно было бы представить таким образцом URL:

```
urlpatterns = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', views.add_stage),
    # ...
)
```

Он совпадает, например, с URL `/myblog/entries/add/` и `/auth/groups/add/`. Однако страница добавления объекта учетной записи пользователя (`/auth/user/add/`) – особый случай, так как на ней отображаются два

поля ввода пароля. Эту проблему *можно было бы* решить, реализовав логику обработки особого случая в самом представлении:

```
def add_stage(request, app_label, model_name):
    if app_label == 'auth' and model_name == 'user':
        # обработка особого случая
    else:
        # обработка обычного случая
```

Однако это неэлегантно по той же причине, которая уже несколько раз упоминалась в этой главе: информация о структуре URL проникает в представление. Правильнее воспользоваться тем, что образцы URL в конфигурации просматриваются сверху вниз:

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', views.user_add_stage),
    ('^([^/]+)/([^/]+)/add/$', views.add_stage),
    # ...
)
```

При такой организации списка запрос к `/auth/user/add/` будет обработан представлением `user_add_stage`. Хотя этот URL соответствует обоим образцам, для его обработки будет вызвано первое представление, так как совпадение с первым образцом будет обнаружено раньше (такая логика называется «сокращенный порядок вычисления»).

Обработка сохраняемых фрагментов текста

Каждый сохраняемый аргумент передается представлению в виде обычной Unicode-строки вне зависимости от его особенностей. Например, при сопоставлении со следующим образцом аргумент `year` будет передан представлению `views.year_archive()` как строка, а не как целое число, несмотря на то что выражение `\d{4}` совпадает только со строками, состоящими из одних цифр:

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

Об этом важно помнить при написании кода представлений. Многие встроенные функции языка Python принимают объекты строго определенного типа (и это правильно). Типичная ошибка – пытаться создать объект `datetime.date`, передав конструктору строки вместо целых чисел:

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
    ...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

В применении к конфигурации URL и представлениям эта ошибка проявится в следующей ситуации:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(\d{4})/(\d{2})/(\d{2})/$', views.day_archive),
)

# views.py

import datetime

def day_archive(request, year, month, day):
    # В следующей инструкции возникнет исключение TypeError!
    date = datetime.date(year, month, day)
```

Корректный код `day_archive()` выглядит так:

```
def day_archive(request, year, month, day):
    date = datetime.date(int(year), int(month), int(day))
```

Отметим, что сама функция `int()` возбуждает исключение `ValueError`, если ей передается строка, содержащая что-то, кроме цифр, но в данном случае этого не произойдет, потому что регулярное выражение в образце URL написано так, что функции представления передается строка, состоящая из одних цифр.

Что сопоставляется с образцами URL

При получении запроса Django пытается сопоставить перечисленные в конфигурации URL образцы с адресом URL запроса, который интерпретируется как строка Python. При сопоставлении не принимаются во внимание ни параметры `GET` и `POST`, ни доменное имя. Также игнорируется символ слеша в начале, потому что он присутствует в любом URL.

Например, при обращении к URL *http://www.example.com/myapp/* Django будет сопоставлять с образцами строку `myapp/`, так же как и при обращении к URL *http://www.example.com/myapp/?page=3*.

Метод отправки запроса (например, `POST` или `GET`) *не* учитывается при сопоставлении. Иными словами, независимо от метода отправки запроса он будет передан для обработки одной и той же функции, которая сама должна организовать ветвление по методу запроса.

Высокоуровневые абстракции функций представления

Раз уж мы заговорили о ветвлении по методу запроса, покажем, как это можно элегантно осуществить. Рассмотрим следующую строку в конфигурации URL:

```
# urls.py

from django.conf.urls.defaults import *
```

```
from mysite import views

urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.some_page),
    # ...
)

# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def some_page(request):
    if request.method == 'POST':
        do_something_for_post()
        return HttpResponseRedirect('/someurl/')
    elif request.method == 'GET':
        do_something_for_get()
        return render_to_response('page.html')
    else:
        raise Http404()
```

В этом примере функция `some_page()` по-разному реагирует на запросы, отправленные методами `POST` и `GET`. Общий у них только URL: `/somepage/`. Однако обрабатывать методы `POST` и `GET` в одной функции не совсем правильно. Куда лучше было бы определить разные функции представления для обработки GET- и POST-запросов соответственно и вызывать ту, которая необходима в конкретном случае.

Это можно сделать, написав функцию представления, которая делегирует содержательную работу другим функциям до или после выполнения некоторых общих действий. Вот как применение этого приема позволяет упростить представление `some_page()`:

```
# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def method_splitter(request, GET=None, POST=None):
    if request.method == 'GET' and GET is not None:
        return GET(request)
    elif request.method == 'POST' and POST is not None:
        return POST(request)
    raise Http404

def some_page_get(request):
    assert request.method == 'GET'
    do_something_for_get()
    return render_to_response('page.html')

def some_page_post(request):
    assert request.method == 'POST'
```

```
        do_something_for_post()
        return HttpResponseRedirect('/someurl/')

# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.method_splitter,
    {'GET': views.some_page_get, 'POST': views.some_page_post}),
    # ...
)
```

Разберемся, что здесь происходит.

- Мы написали новую функцию представления `method_splitter()`, которая делегирует работу другим представлениям в зависимости от значения `request.method`. Она ожидает получить два именованных аргумента, `GET` и `POST`, которые обязаны быть *функциями представлений*. Если значение `request.method` равно `'GET'`, то вызывается функция `GET`. Если значение `request.method` равно `'POST'`, то вызывается функция `POST`. Если же значение `request.method` равно чему-то еще (например, `HEAD`) или ожидаемый аргумент (соответственно `GET` или `POST`) не был передан, то возбуждается исключение `Http404`.
- В конфигурации URL мы ассоциируем с образцом `/somepage/` представление `method_splitter()` и передаем ему дополнительные аргументы: функции представлений, которые нужно вызывать для запросов типа GET и POST соответственно.
- Наконец, функция `some_page()` разбивается на две: `some_page_get()` и `some_page_post()`. Это гораздо элегантнее, чем помещать всю логику обработки в одно представление.

Примечание

Строго говоря, новые функции представления не обязаны проверять значение `request.method`, так как это уже сделала функция `method_splitter()`. (Гарантируется, что в момент вызова `some_page_post()` атрибут `request.method` равен `'POST'`.) Но на всякий случай мы все-таки добавили утверждение `assert`, проверяющее, что получен ожидаемый аргумент. Заодно это утверждение документирует назначение функции.

Теперь у нас есть обобщенная функция представления, которая инкапсулирует логику ветвления по значению атрибута `request.method`. В функции `method_splitter()` нет ничего специфичного для конкретного приложения, поэтому ее можно использовать и в других проектах.

Однако функцию `method_splitter()` можно немного улучшить. Сейчас предполагается, что представлениям, обрабатывающим GET- и POST-запросы, не передается никаких аргументов, кроме `request`. А если мы

захотим использовать method_splitter() в сочетании с представлениями, которым, к примеру, нужны какие-то части URL или которые принимают дополнительные именованные аргументы? Что тогда?

Для решения этой проблемы можно воспользоваться удобной возможностью Python: списками аргументов переменной длины, обозначаемыми звездочкой. Сначала приведем пример, а потом дадим пояснения:

```
def method_splitter(request, *args, **kwargs):
    get_view = kwargs.pop('GET', None)
    post_view = kwargs.pop('POST', None)
    if request.method == 'GET' and get_view is not None:
        return get_view(request, *args, **kwargs)
    elif request.method == 'POST' and post_view is not None:
        return post_view(request, *args, **kwargs)
    raise Http404
```

Мы переделали функцию method_splitter(), заменив именованные аргументы GET и POST на *args и **kwargs (обратите внимание на звездочки). Этот механизм позволяет функции принимать переменное количество аргументов, имена которых неизвестны до момента выполнения. Если поставить одну звездочку перед именем параметра в определении функции, то все *позиционные* аргументы будут помещены в один кортеж. Если же перед именем параметра стоят две звездочки, то все *именованные* аргументы помещаются в один словарь.

Рассмотрим, к примеру, такую функцию:

```
def foo(*args, **kwargs):
    print "Позиционные аргументы:"
    print args
    print "Именованные аргументы:"
    print kwargs
```

Работает она следующим образом:

```
>>> foo(1, 2, 3)
Позиционные аргументы:
(1, 2, 3)
Именованные аргументы:
{}
>>> foo(1, 2, name='Adrian', framework='Django')
Позиционные аргументы:
(1, 2)
Именованные аргументы:
{'framework': 'Django', 'name': 'Adrian'}
```

Но вернемся к функции method_splitter(). Как теперь стало понятно, использование *args и **kwargs позволяет ей принимать *любые* аргументы и передавать их дальше соответствующему представлению. Но предварительно мы дважды обращаемся к методу kwargs.pop(), чтобы получить аргументы GET и POST, если они присутствуют в словаре. (Мы вызываем

pop()со значением по умолчанию None, чтобы избежать ошибки KeyError в случае, когда искомый ключ отсутствует в словаре.)

Обертывание функций представления

Последний прием, который мы рассмотрим, опирается на относительно редко используемую возможность Python. Предположим, что в разных представлениях многократно встречается один и тот же код, например:

```
def my_view1(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template1.html')

def my_view2(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template2.html')

def my_view3(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template3.html')
```

Здесь в начале каждого представления проверяется, что пользователь request.user аутентифицирован, то есть успешно прошел процедуру проверки при заходе на сайт. Если это не так, производится переадресация на страницу /accounts/login/.

Примечание

Мы еще не говорили об объекте request.user – это тема главы 14, – но отметим, что request.user представляет текущего пользователя, аутентифицированного или анонимного.

Хорошо бы убрать повторяющийся код и просто как-то пометить, что представления требуют аутентификации. Это можно сделать с помощью обертки представления. Взгляните на следующий фрагмент:

```
def requires_login(view):
    def new_view(request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect('/accounts/login/')
        return view(request, *args, **kwargs)
    return new_view
```

Функция requires_login принимает функцию представления (view) и возвращает новую функцию представления (new_view). Функция new_view определена *внутри* requires_login, она проверяет request.user.

is_authenticated() и делегирует работу исходному представлению view. Теперь можно убрать проверки if not request.user.is_authenticated()из наших представлений и просто обернуть их функцией requires_login в конфигурации URL:

```
from django.conf.urls.defaults import *
from mysite.views import requires_login, my_view1, my_view2, my_view3

urlpatterns = patterns('',
    (r'^view1/$', requires_login(my_view1)),
    (r'^view2/$', requires_login(my_view2)),
    (r'^view3/$', requires_login(my_view3)),
)
```

Результат при этом не изменяется, а дублирование кода устранено. Имея обобщенную функцию requires_login(), мы можем обернуть ею любое представление, которое требует предварительной аутентификации.

Включение других конфигураций URL

Если вы планируете использовать свой код на нескольких сайтах, созданных на основе Django, то должны организовать свою конфигурацию URL так, чтобы она допускала возможность включения в другие конфигурации.

Внешние модули конфигурации URL можно включать в любой точке имеющейся конфигурации. Вот пример конфигурации URL, включающей другие конфигурации:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

Мы уже встречались с этим механизмом в главе 6, когда рассматривали административный интерфейс Django. У административного интерфейса имеется собственная конфигурация URL, которую вы включаете с помощью функции include().

Но обратите внимание на одну тонкость: регулярные выражения, указывающие на include(), *не заканчиваются* метасимволом $ (совпадающим с концом строки), зато *заканчиваются* символом слеша. Встретив include(), Django отбрасывает ту часть URL, которая совпала к этому моменту, и передает остаток строки включаемой конфигурации для последующей обработки.

Продолжая тот же пример, приведем конфигурацию mysite.blog.urls:

```
from django.conf.urls.defaults import *
```

```
    urlpatterns = patterns('',
        (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
        (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

Теперь покажем, как при наличии таких двух конфигураций обрабатываются некоторые запросы.

- /weblog/2007/: URL совпадает с образцом `r'^weblog/'` из первой конфигурации URL. Поскольку с ним ассоциирован `include()`, Django отбрасывает совпавший текст, в данном случае `'weblog/'`. Остается часть 2007/, которая совпадает с первой строкой в конфигурации `mysite.blog.urls`.
- /weblog//2007/ (с двумя символами слеша): URL совпадает с образцом `r'^weblog/'` из первой конфигурации URL. Поскольку с ним ассоциирован `include()`, Django отбрасывает совпавший текст, в данном случае `'weblog/'`. Остается часть /2007/ (с символом слеша в начале), которая не совпадает ни с одной строкой в конфигурации `mysite.blog.urls`.
- /about/: URL совпадает с образцом `mysite.views.about` в первой конфигурации URL; это доказывает, что в одной и той же конфигурации могут употребляться образцы, содержащие и не содержащие `include()`.

Сохраняемые параметры и механизм include()

Включаемая конфигурация URL получает все сохраненные параметры из родительской конфигурации, например:

```
# root urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# foo/urls/blog.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'foo.views.blog_index'),
    (r'^archive/$', 'foo.views.blog_archive'),
)
```

В этом примере сохраняемая переменная `username` передается во включаемую конфигурацию URL и, следовательно, *всем* функциям представления, упоминаемым в этой конфигурации.

Отметим, что сохраняемые параметры *всегда* передаются *каждой* строке включаемой конфигурации вне зависимости от того, допустимы ли они для указанной в этой строке функции представления. Поэтому опи-

санная техника полезна лишь в том случае, когда вы уверены, что любое представление, упоминаемое во включаемой конфигурации, принимает передаваемые параметры.

Дополнительные параметры в конфигурации URL и механизм include()

Во включаемую конфигурацию URL можно передавать дополнительные параметры обычным образом – в виде словаря. Но при этом дополнительные параметры будут передаваться *каждой* строке включаемой конфигурации.

Например, следующие два набора конфигурации URL функционально эквивалентны.

Первый набор:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

Второй набор:

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
    (r'^rss/$', 'mysite.views.rss', {'blogid': 3}),
)
```

Как и сохраняемые параметры (см. предыдущий раздел), дополнительные параметры *всегда* передаются *каждой* строке включаемой конфигурации вне зависимости от того, допустимы ли они для указанной в этой строке функции представления. Поэтому описанная техника полезна лишь в том случае, когда вы уверены, что любое представление, упоминаемое во включаемой конфигурации, принимает дополнительные параметры, которые ей передаются.

Что дальше?

В этой главе приведено много полезных приемов работы с представлениями и конфигурациями URL. В главе 9 мы столь же углубленно рассмотрим систему шаблонов в Django.