The Django template language¶

About this document

This document explains the language syntax of the Django template system. If you're looking for a more technical perspective on how it works and how to extend it, see The Django template language: For Python programmers.

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as Smarty or CheetahTemplate, you should feel right at home with Django's templates.

Philosophy

If you have a background in programming, or if you're used to languages which mix programming code directly into HTML, you'll want to bear in mind that the Django template system is not simply Python embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.

The Django template system provides tags which function similarly to some programming constructs – an if tag for boolean tests, a for tag for looping, etc. – but these are not simply executed as the corresponding Python code, and the template system will not execute arbitrary Python expressions. Only the tags, filters and syntax listed below are supported by default (although you can add your own extensions to the template language as needed).

Templates

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
 <a href="{{ story.get_absolute_url }}">
  {{ story.headline|upper }}
 </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Philosophy

Why use a text-based template instead of an XML-based one (like Zope's TAL)? We wanted Django's template language to be usable for more than just XML/HTML templates. At World Online, we use it for emails, JavaScript and CSV. You can use the template language for any text-based format.

Oh, and one more thing: Making humans edit XML is sadistic!

Variables

Variables look like this: {{ variable }}. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore ("_"). The dot (".") also appears in variable sections, although that has a special meaning, as indicated below. Importantly, you cannot have spaces or punctuation characters in

variable names.
Use a dot (.) to access attributes of a variable.
Behind the scenes
Technically, when the template system encounters a dot, it tries the following lookups, in this order:
Dictionary lookup
Attribute lookup
Method call
List-index lookup
This can cause some unexpected behavior with objects that override dictionary lookup. For example, consider the following code snippet that attempts to loop over a collections.defaultdict:

```
{% for k, v in defaultdict.iteritems %}
    Do something with k and v here...
{% endfor %}
```

Because dictionary lookup happens first, that behavior kicks in and provides a default value instead of using the intended.iteritems() method. In this case, consider converting to a dictionary first.
In the above example, {{ section.title }} will be replaced with the title attribute of the section object.
If you use a variable that doesn't exist, the template system will insert the value of the TEMPLATE_STRING_IF_INVALID setting, which is set to "(the empty string) by default.
Note that "bar" in a template expression like {{ foo.bar }} will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context.

Filters

You can modify variables for display by using filters.
Filters look like this: {{ name|lower }}. This displays the value of the {{ name }} variable after being filtered through the lower filter, which converts text to lowercase. Use a pipe (|) to apply a filter.
Filters can be "chained." The output of one filter is applied to the next. {{ text|escape|linebreaks }} is a common idiom for escaping text contents, then converting line breaks to <p> tags.
Some filters take arguments. A filter argument looks like this: {{ bio|truncatewords:30 }}. This will display the first 30 words of the biovariable.
Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaced you'd use {{ list|join:", " }}.
Django provides about thirty built-in template filters. You can read all about them in the built-in filter reference. To give you a taste of what's available, here are some of the more commonly used template filters:

default

If a variable is false or empty, use given default. Otherwise, use the value of the variable
For example:

```
{{ value|default:"nothing" }}
```

If value isn't provided or is empty, the above will display "nothing".

length

Returns the length of the value. This works for both strings and lists; for example:

```
{{ value|length }}
```

If value is ['a', 'b', 'c', 'd'], the output will be 4.

striptags

Strips all [X]HTML tags. For example:

```
{{ value|striptags }}
```

If value is "Joel <button>is</button> a slug", the output will be "Joel is a slug".
Again, these are just a few examples; see the built-in filter reference for the complete list.
You can also create your own custom template filters; see Custom template tags and filters.
See also
Django's admin interface can include a complete reference of all template tags and filters available for a given site. See The Django admin documentation generator.

Tags

Tags look like this: {% tag %}. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. {% tag %} ... tag contents ... {% endtag %}).

Django ships with about two dozen built-in template tags. You can read all about them in the built-in tag reference. To give you a taste of what's available, here are some of the more commonly used tags:

for

Loop over each item in an array. For example, to display a list of athletes provided in athlete_list:

```
<ul>
{% for athlete in athlete_list %}
  <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

if and else

Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
  Number of athletes: {{ athlete_list|length }}
{% else %}
  No athletes.
{% endif %}
```

In the above, if athlete_list is not empty, the number of athletes will be displayed by the {{ athlete_list|length }} variable.

You can also use filters and various operators in the if tag:

```
{% if athlete_list|length > 1 %}
  Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
  Athlete: {{ athlete_list.0.name }}
{% endif %}
```

While the above example works, be aware that most template filters return strings, so mathematical comparisons using filters will generally not work as you expect. length is an exception.

block and extends

Set up template inheritance (see below), a powerful way of cutting down on "boilerplate" in templates.

Again, the above is only a selection of the whole list; see the built-in tag reference for the complete list.

You can also create your own custom template tags; see Custom template tags and filters.

See also

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See The Django admin documentation generator.

Comments

To comment-out part of a line in a template, use the comment syntax: {# #}.

For example, this template would render as 'hello':

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the {# and #} delimiters). If you need to comment out a multiline portion of the template, see the comment tag.

Template inheritance

The most powerful – and thus the most complex – part of Django's template engine is template inheritance.

Template inheritance allows you to build a base "skeleton" template that contains all the common elements

of your site and defines blocks that child templates can override.
It’s easiest to understand template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="style.css" />
  <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
  <div id="sidebar">
    {% block sidebar %}
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/blog/">Blog</a></li>
    </ul>
    {% endblock %}
  </div>

  <div id="content">
    {% block content %}{% endblock %}
  </div>
</body>
</html>
```

This template, which we’ll call base.html, defines a simple HTML skeleton document that you might use for a simple two-column page. It’s the job of “child” templates to fill the empty blocks with content.
In this example, the block tag defines three blocks that child templates can fill in. All the block tag does is to tell the template engine that a child template may override those portions of the template.
A child template might look like this:

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
  <h2>{{ entry.title }}</h2>
  <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

The extends tag is the key here. It tells the template engine that this template “extends” another template. When the template system evaluates this template, first it locates the parent – in this case, “base.html”.
At that point, the template engine will notice the three block tags in base.html and replace those blocks with the contents of the child template. Depending on the value of blog_entries, the output might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="style.css" />
  <title>My amazing blog</title>
</head>

<body>
  <div id="sidebar">
```

```
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

Note that since the child template didn't define the sidebar block, the value from the parent template is used instead. Content within a {% block %} tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

Create a base.html template that holds the main look-and-feel of your site.

Create a base_SECTIONNAME.html template for each "section" of your site. For example, base_news.html, base_sports.html. These templates all extend base.html and include section-specific styles/design.

Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

If you use {% extends %} in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.

More {% block %} tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.

If you find yourself duplicating content in a number of templates, it probably means you should move that content to a {% block %} in a parent template.

If you need to get the content of the block from the parent template, the {{ block.super }} variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using {{ block.super }} will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.

For extra readability, you can optionally give a name to your {% endblock %} tag. For example:

```
{% block content %}
...
{% endblock content %}
```

In larger templates, this technique helps you see which {% block %} tags are being closed.

Finally, note that you can't define multiple block tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill – it also defines the content that fills the hole in the parent. If there were two similarly-named block tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

Automatic HTML escaping

When generating HTML from templates, there's always a risk that a variable will include characters that

affect the resulting HTML. For example, consider this template fragment:
Hello, {{ name }}.
At first, this seems like a harmless way to display a user's name, but consider what would happen if the user entered his name as this:
<script>alert('hello')</script>
With this name value, the template would be rendered as:
Hello, <script>alert('hello')</script>
...which means the browser would pop-up a JavaScript alert box!
Similarly, what if the name contained a '<' symbol, like this?
<b>username
That would result in a rendered template like this:
Hello, <b>username
...which, in turn, would result in the remainder of the Web page being bolded!
Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your Web pages, because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a Cross Site Scripting (XSS) attack.
To avoid this problem, you have two options:
One, you can make sure to run each untrusted variable through the escape filter (documented below), which converts potentially harmful HTML characters to unharmful ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus onyou, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
Two, you can take advantage of Django's automatic HTML escaping. The remainder of this section describes how auto-escaping works.
By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:
< is converted to <
> is converted to >
' (single quote) is converted to '
" (double quote) is converted to "
& is converted to &
Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

How to turn it off

If you don't want data to be auto-escaped, on a per-site, per-template level or per-variable level, you can turn it off in several ways.
Why would you want to turn it off? Because sometimes, template variables contain data that you intend to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of HTML in your database and want to embed that directly into your template. Or, you might be using Django's template system to produce text that is not HTML – like an email message, for instance.

For individual variables

To disable auto-escaping for an individual variable, use the safe filter:
This will be escaped: {{ data }}
This will not be escaped: {{ data|**safe** }}
Think of safe as shorthand for safe from further escaping or can be safely interpreted as HTML. In this example, if data contains '<b>', the output will be:
This will be escaped: <b>
This will not be escaped: <b>

For template blocks

To control auto-escaping for a template, wrap the template (or just a particular section of the template) in

the autoescape tag, like so:

```
{% autoescape off %}
  Hello {{ name }}
{% endautoescape %}
```

The autoescape tag takes either on or off as its argument. At times, you might want to force auto-escaping when it would otherwise be disabled. Here is an example template:

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
  This will not be auto-escaped: {{ data }}.
  Nor this: {{ other_data }}
  {% autoescape on %}
    Auto-escaping applies again: {{ name }}
  {% endautoescape %}
{% endautoescape %}
```

The auto-escaping tag passes its effect onto templates that extend the current one as well as templates included via the include tag, just like all block tags. For example:

```
# base.html
{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
# child.html
{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the greeting variable contains the string Hello!:

```
<h1>This & that</h1>
<b>Hello!</b>
```

Notes

Generally, template authors don’t need to worry about auto-escaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn’t be escaped, and mark data appropriately, so things Just Work in the template.

If you’re creating a template that might be used in situations where you’re not sure whether auto-escaping is enabled, then add an escape filter to any variable that needs escaping. When auto-escaping is on, there’s no danger of the escape filter double-escaping data – the escape filter does not affect auto-escaped variables.

String literals and automatic escaping

As we mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted without any automatic escaping into the template – they act as if they were all passed through the safe filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write

```
{{ data|default:"3 &lt; 2" }}
```

...rather than

```
{{ data|default:"3 < 2" }}  <-- Bad! Don't do this.
```

This doesn’t affect what happens to data coming from the variable itself. The variable’s contents are still automatically escaped, if necessary, because they’re beyond the control of the template author.

Accessing method calls

Most method calls attached to objects are also available from within templates. This means that templates have access to much more than just class attributes (like field names) and variables passed in from views. For example, the Django ORM provides the "entry_set" syntax for finding a collection of objects related on a foreign key. Therefore, given a model called "comment" with a foreign key relationship to a model called "task" you can loop through all comments attached to a given task like this:

```
{% for comment in task.comment_set.all %}
  {{ comment }}
{% endfor %}
```

Similarly, QuerySets provide a count() method to count the number of objects they contain. Therefore, you can obtain a count of all comments related to the current task with:

```
{{ task.comment_set.all.count }}
```

And of course you can easily access methods you've explicitly defined on your own models:

```
# In model
class Task(models.Model):
  def foo(self):
    return "bar"
# In template
{{ task.foo }}
```

Because Django intentionally limits the amount of logic processing available in the template language, it is not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views, then passed to templates for display.

Custom tag and filter libraries

Certain applications provide custom tag and filter libraries. To access them in a template, use the load tag:

```
{% load comments %}
{% comment_form for blogs.entries entry.id with is_public yes %}
```

In the above, the load tag loads the comments tag library, which then makes the comment_form tag available for use. Consult the documentation area in your admin to find the list of custom libraries in your installation.

The load tag can take multiple library names, separated by spaces. Example:

```
{% load comments i18n %}
```

See Custom template tags and filters for information on writing your own custom template libraries.

Custom libraries and template inheritance

When you load a custom tag or filter library, the tags/filters are only made available to the current template – not any parent or child templates along the template-inheritance path.

For example, if a template foo.html has {% load comments %}, a child template (e.g., one that has {% extends "foo.html" %}) will nothave access to the comments template tags and filters. The child template is responsible for its own {% load comments %}.

This is a feature for the sake of maintainability and sanity.

The Django template language: For Python programmers¶

This document explains the Django template system from a technical perspective – how it works and how to extend it. If you're just looking for reference on the language syntax, see The Django template language.

If you're looking to use the Django template system as part of another application – i.e., without the rest of the framework – make sure to read theconfiguration section later in this document.

Basics

A template is a text document, or a normal Python string, that is marked-up using the Django template language. A template can contain block tags or variables.

A block tag is a symbol within a template that does something.

This definition is deliberately vague. For example, a block tag can output content, serve as a control

structure (an “if” statement or “for” loop), grab content from a database or enable access to other template tags.
Block tags are surrounded by "{%" and "%}".
Example template with block tags:
{% **if** is_logged_in %}Thanks for logging in!{% **else** %}Please log in.{% **endif** %}
A variable is a symbol within a template that outputs a value.
Variable tags are surrounded by "{{" and "}}".
Example template with variables:
My first name is {{ first_name }}. My last name is {{ last_name }}.
A context is a “variable name” -> “variable value” mapping that is passed to a template.
A template renders a context by replacing the variable “holes” with values from the context and executing all block tags.

Using the template system

class Template

Using the template system in Python is a two-step process:
First, you compile the raw template code into a Template object.
Then, you call the render() method of the Template object with a given context.

Compiling a string

The easiest way to create a Template object is by instantiating it directly. The class lives at django.template.Template. The constructor takes one argument – the raw template code:

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print(t)
<django.template.Template instance>
```

Behind the scenes

The system only parses your raw template code once – when you create the Template object. From then on, it’s stored internally as a “node” structure for performance.
Even the parsing itself is quite fast. Most of the parsing happens via a single call to a single, short, regular expression.

Rendering a context

render(context)

Once you have a compiled Template object, you can render a context – or multiple contexts – with it. The Context class lives atdjango.template.Context, and the constructor takes two (optional) arguments:
A dictionary mapping variable names to variable values.
The name of the current application. This application name is used to help resolve namespaced URLs. If you’re not using namespaced URLs, you can ignore this argument.
Call the Template object’s render() method with the context to “fill” the template:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ my_name }}.")

>>> c = Context({"my_name": "Adrian"})
>>> t.render(c)
"My name is Adrian."

>>> c = Context({"my_name": "Dolores"})
>>> t.render(c)
"My name is Dolores."
```

Variables and lookups

Variable names must consist of any letter (A-Z), any digit (0-9), an underscore (but they must not start with

an underscore) or a dot.
Dots have a special meaning in template rendering. A dot in a variable name signifies a lookup. Specifically, when the template system encounters a dot in a variable name, it tries the following lookups, in this order:
Dictionary lookup. Example: foo["bar"]
Attribute lookup. Example: foo.bar
List-index lookup. Example: foo[bar]
Note that “bar” in a template expression like {{ foo.bar }} will be interpreted as a literal string and not using the value of the variable “bar”, if one exists in the template context.
The template system uses the first lookup type that works. It’s short-circuit logic. Here are a few examples:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ person.first_name }}.")
>>> d = {"person": {"first_name": "Joe", "last_name": "Johnson"}}
>>> t.render(Context(d))
"My name is Joe."

>>> class PersonClass: pass
>>> p = PersonClass()
>>> p.first_name = "Ron"
>>> p.last_name = "Nasty"
>>> t.render(Context({"person": p}))
"My name is Ron."

>>> t = Template("The first stooge in the list is {{ stooges.0 }}.")
>>> c = Context({"stooges": ["Larry", "Curly", "Moe"]})
>>> t.render(c)
"The first stooge in the list is Larry."
```

If any part of the variable is callable, the template system will try calling it. Example:

```
>>> class PersonClass2:
...     def name(self):
...         return "Samantha"
>>> t = Template("My name is {{ person.name }}.")
>>> t.render(Context({"person": PersonClass2}))
"My name is Samantha."
```

Callable variables are slightly more complex than variables which only require straight lookups. Here are some things to keep in mind:
If the variable raises an exception when called, the exception will be propagated, unless the exception has an attributesilent_variable_failure whose value is True. If the exception does have a silent_variable_failure attribute whose value is True, the variable will render as an empty string. Example:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError("foo")
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(Exception):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
```

```
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

Note that django.core.exceptions.ObjectDoesNotExist, which is the base class for all Django database API DoesNotExistexceptions, has silent_variable_failure = True. So if you're using Django templates with Django model objects, any DoesNotExistexception will fail silently.

A variable can only be called if it has no required arguments. Otherwise, the system will return an empty string.

Obviously, there can be side effects when calling some variables, and it'd be either foolish or a security hole to allow the template system to access them.

A good example is the delete() method on each Django model object. The template system shouldn't be allowed to do something like this:

I will now delete this valuable data. {{ data.delete }}

To prevent this, set an alters_data attribute on the callable variable. The template system won't call a variable if it has alters_data=Trueset, and will instead replace the variable with TEMPLATE_STRING_IF_INVALID, unconditionally. The dynamically-generated delete() andsave() methods on Django model objects get alters_data=True automatically. Example:

```
def sensitive_function(self):
    self.database_record.delete()
sensitive_function.alters_data = True
```

Occasionally you may want to turn off this feature for other reasons, and tell the template system to leave a variable un-called no matter what. To do so, set a do_not_call_in_templates attribute on the callable with the value True. The template system then will act as if your variable is not callable (allowing you to access attributes of the callable, for example).

How invalid variables are handled

Generally, if a variable doesn't exist, the template system inserts the value of the TEMPLATE_STRING_IF_INVALID setting, which is set to '' (the empty string) by default.

Filters that are applied to an invalid variable will only be applied if TEMPLATE_STRING_IF_INVALID is set to '' (the empty string). IfTEMPLATE_STRING_IF_INVALID is set to any other value, variable filters will be ignored.

This behavior is slightly different for the if, for and regroup template tags. If an invalid variable is provided to one of these template tags, the variable will be interpreted as None. Filters are always applied to invalid variables within these template tags.

If TEMPLATE_STRING_IF_INVALID contains a '%s', the format marker will be replaced with the name of the invalid variable.

For debug purposes only!

While TEMPLATE_STRING_IF_INVALID can be a useful debugging tool, it is a bad idea to turn it on as a 'development default'.

Many templates, including those in the Admin site, rely upon the silence of the template system when a non-existent variable is encountered. If you assign a value other than '' to TEMPLATE_STRING_IF_INVALID, you will experience rendering problems with these templates and sites.

Generally, TEMPLATE_STRING_IF_INVALID should only be enabled in order to debug a specific template problem, then cleared once debugging is complete.

Builtin variables

Every context contains True, False and None. As you would expect, these variables resolve to the corresponding Python objects.

New in Django 1.5:

Before Django 1.5, these variables weren't a special case, and they resolved to None unless you defined

them in the context.

Playing with Context objects

class Context

Most of the time, you'll instantiate Context objects by passing in a fully-populated dictionary to Context(). But you can add and delete items from a Context object once it's been instantiated, too, using standard dictionary syntax:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

Context.pop()

Context.push()

exception ContextPopException

A Context object is a stack. That is, you can push() and pop() it. If you pop() too much, it'll raise django.template.ContextPopException:

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.push()
>>> c['foo'] = 'second level'
>>> c['foo']
'second level'
>>> c.pop()
>>> c['foo']
'first level'
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'
>>> c.pop()
Traceback (most recent call last):
...
django.template.ContextPopException
```

update(other_dict)

In addition to push() and pop(), the Context object also defines an update() method. This works like push() but takes a dictionary as an argument and pushes that dictionary onto the stack instead of an empty one.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.update({'foo': 'updated'})
{'foo': 'updated'}
>>> c['foo']
'updated'
>>> c.pop()
{'foo': 'updated'}
>>> c['foo']
'first level'
```

Using a Context as a stack comes in handy in some custom template tags, as you'll see below.

Subclassing Context: RequestContext

class RequestContext

Django comes with a special Context class, django.template.RequestContext, that acts slightly differently than the normaldjango.template.Context. The first difference is that it takes an HttpRequest as its first argument. For example:

```
c = RequestContext(request, {
    'foo': 'bar',
})
```

The second difference is that it automatically populates the context with a few variables, according to your TEMPLATE_CONTEXT_PROCESSORSsetting.

The TEMPLATE_CONTEXT_PROCESSORS setting is a tuple of callables – called context processors – that take a request object as their argument and return a dictionary of items to be merged into the context. By default, TEMPLATE_CONTEXT_PROCESSORS is set to:

```
("django.contrib.auth.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media",
"django.core.context_processors.static",
"django.core.context_processors.tz",
"django.contrib.messages.context_processors.messages")
```

In addition to these, RequestContext always uses django.core.context_processors.csrf. This is a security related context processor required by the admin and other contrib apps, and, in case of accidental misconfiguration, it is deliberately hardcoded in and cannot be turned off by the TEMPLATE_CONTEXT_PROCESSORS setting.

Each processor is applied in order. That means, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first. The default processors are explained below.

When context processors are applied

When you use RequestContext, the variables you supply directly are added first, followed any variables supplied by context processors. This means that a context processor may overwrite a variable you've supplied, so take care to avoid variable names which overlap with those supplied by your context processors.

Also, you can give RequestContext a list of additional processors, using the optional, third positional argument, processors. In this example, the RequestContext instance gets a ip_address variable:

```
from django.http import HttpResponse
from django.template import RequestContext

def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

def some_view(request):
    # ...
    c = RequestContext(request, {
        'foo': 'bar',
    }, [ip_address_processor])
    return HttpResponse(t.render(c))
```

Note

If you're using Django's render_to_response() shortcut to populate a template with the contents of a dictionary, your template will be passed a Context instance by default (not a RequestContext). To use

a RequestContext in your template rendering, pass an optional third argument to render_to_response(): a RequestContext instance. Your code might look like this:

```
from django.shortcuts import render_to_response
from django.template import RequestContext

def some_view(request):
    # ...
    return render_to_response('my_template.html',
                              my_data_dictionary,
                              context_instance=RequestContext(request))
```

Alternatively, use the render() shortcut which is the same as a call to render_to_response() with a context_instance argument that forces the use of a RequestContext.

Here's what each of the default processors does:

django.contrib.auth.context_processors.auth

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain these variables:

user – An auth.User instance representing the currently logged-in user (or an AnonymousUser instance, if the client isn't logged in).

perms – An instance of django.contrib.auth.context_processors.PermWrapper, representing the permissions that the currently logged-in user has.

django.core.context_processors.debug

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain these two variables – but only if your DEBUGsetting is set to True and the request's IP address (request.META['REMOTE_ADDR']) is in the INTERNAL_IPS setting:

debug – True. You can use this in templates to test whether you're in DEBUG mode.

sql_queries – A list of {'sql': ..., 'time': ...} dictionaries, representing every SQL query that has happened so far during the request and how long it took. The list is in order by query.

django.core.context_processors.i18n

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain these two variables:

LANGUAGES – The value of the LANGUAGES setting.

LANGUAGE_CODE – request.LANGUAGE_CODE, if it exists. Otherwise, the value of the LANGUAGE_CODE setting.

See Internationalization and localization for more.

django.core.context_processors.media

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a variable MEDIA_URL, providing the value of theMEDIA_URL setting.

django.core.context_processors.static

static()

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a variable STATIC_URL, providing the value of the STATIC_URL setting.

django.core.context_processors.csrf

This processor adds a token that is needed by the csrf_token template tag for protection against Cross Site Request Forgeries.

django.core.context_processors.request

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a variable request, which is the currentHttpRequest. Note that this processor is not enabled by default; you'll have to activate it.

django.contrib.messages.context_processors.messages

If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a single additional variable:

messages – A list of messages (as strings) that have been set via the user model

(using user.message_set.create) or through themessages framework.

Writing your own context processors

A context processor has a very simple interface: It's just a Python function that takes one argument, an HttpRequest object, and returns a dictionary that gets added to the template context. Each context processor must return a dictionary.

Custom context processors can live anywhere in your code base. All Django cares about is that your custom context processors are pointed-to by your TEMPLATE_CONTEXT_PROCESSORS setting.

Loading templates

Generally, you'll store templates in files on your filesystem rather than using the low-level Template API yourself. Save templates in a directory specified as a template directory.

Django searches for template directories in a number of places, depending on your template-loader settings (see "Loader types" below), but the most basic way of specifying template directories is by using the TEMPLATE_DIRS setting.

The TEMPLATE_DIRS setting

Tell Django what your template directories are by using the TEMPLATE_DIRS setting in your settings file. This should be set to a list or tuple of strings that contain full paths to your template directory(ies). Example:

```
TEMPLATE_DIRS = (
    "/home/html/templates/lawrence.com",
    "/home/html/templates/default",
)
```

Your templates can go anywhere you want, as long as the directories and templates are readable by the Web server. They can have any extension you want, such as .html or .txt, or they can have no extension at all.

Note that these paths should use Unix-style forward slashes, even on Windows.

The Python API

django.template.loader has two functions to load templates from files:

get_template(template_name)

get_template returns the compiled template (a Template object) for the template with the given name. If the template doesn't exist, it raisesdjango.template.TemplateDoesNotExist.

select_template(template_name_list)

select_template is just like get_template, except it takes a list of template names. Of the list, it returns the first template that exists.

For example, if you call get_template('story_detail.html') and have the above TEMPLATE_DIRS setting, here are the files Django will look for, in order:

/home/html/templates/lawrence.com/story_detail.html
/home/html/templates/default/story_detail.html

If you call select_template(['story_253_detail.html', 'story_detail.html']), here's what Django will look for:

/home/html/templates/lawrence.com/story_253_detail.html
/home/html/templates/default/story_253_detail.html
/home/html/templates/lawrence.com/story_detail.html
/home/html/templates/default/story_detail.html

When Django finds a template that exists, it stops looking.

Tip

You can use select_template() for super-flexible "templatability." For example, if you've written a news story and want some stories to have custom templates, use something likeselect_template(['story_%s_detail.html' % story.id, 'story_detail.html']). That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

Using subdirectories

It's possible – and preferable – to organize templates in subdirectories of the template directory. The

convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, just use a slash, like so:

```
get_template('news/story_detail.html')
```

Using the same TEMPLATE_DIRS setting from above, this example get_template() call will attempt to load the following templates:

/home/html/templates/lawrence.com/news/story_detail.html
/home/html/templates/default/news/story_detail.html

Loader types

By default, Django uses a filesystem-based template loader, but Django comes with a few other template loaders, which know how to load templates from other sources.

Some of these other loaders are disabled by default, but you can activate them by editing your TEMPLATE_LOADERS setting. TEMPLATE_LOADERSshould be a tuple of strings, where each string represents a template loader class. Here are the template loaders that come with Django:

django.template.loaders.filesystem.Loader

class filesystem.Loader

Loads templates from the filesystem, according to TEMPLATE_DIRS. This loader is enabled by default.

django.template.loaders.app_directories.Loader

class app_directories.Loader

Loads templates from Django apps on the filesystem. For each app in INSTALLED_APPS, the loader looks for a templates subdirectory. If the directory exists, Django looks for templates in there.

This means you can store templates with your individual apps. This also makes it easy to distribute Django apps with default templates.

For example, for this setting:

```
INSTALLED_APPS = ('myproject.polls', 'myproject.music')
```

...then get_template('foo.html') will look for foo.html in these directories, in this order:

/path/to/myproject/polls/templates/
/path/to/myproject/music/templates/

... and will use the one it finds first.

The order of INSTALLED_APPS is significant! For example, if you want to customize the Django admin, you might choose to override the standard admin/base_site.html template, from django.contrib.admin, with your own admin/base_site.html in myproject.polls. You must then make sure that your myproject.polls comes before django.contrib.admin in INSTALLED_APPS,

otherwisedjango.contrib.admin's will be loaded first and yours will be ignored.

Note that the loader performs an optimization when it is first imported: it caches a list of which INSTALLED_APPS packages have a templatessubdirectory.

This loader is enabled by default.

django.template.loaders.eggs.Loader

class eggs.Loader

Just like app_directories above, but it loads templates from Python eggs rather than from the filesystem.

This loader is disabled by default.

django.template.loaders.cached.Loader

class cached.Loader

By default, the templating system will read and compile your templates every time they need to be rendered. While the Django templating system is quite fast, the overhead from reading and compiling templates can add up.

The cached template loader is a class-based loader that you configure with a list of other loaders that it should wrap. The wrapped loaders are used to locate unknown templates when they are first encountered. The cached loader then stores the compiled Template in memory. The cached Template instance is returned for subsequent requests to load the same template.

For example, to enable template caching with the filesystem and app_directories template loaders you might use the following settings:

```
TEMPLATE_LOADERS = (
    ('django.template.loaders.cached.Loader', (
        'django.template.loaders.filesystem.Loader',
        'django.template.loaders.app_directories.Loader',
    )),
)
```

Note

All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or that you wrote yourself, you should ensure that the Node implementation for each tag is thread-safe. For more information, see template tag thread safety considerations.

This loader is disabled by default.

Django uses the template loaders in order according to the TEMPLATE_LOADERS setting. It uses each loader until a loader finds a match.

The render_to_string shortcut

django.template.loader.render_to_string(template_name, dictionary=None, context_instance=None)

To cut down on the repetitive nature of loading and rendering templates, Django provides a shortcut function which largely automates the process:render_to_string() in django.template.loader, which loads a template, renders it and returns the resulting string:

```
from django.template.loader import render_to_string
rendered = render_to_string('my_template.html', {'foo': 'bar'})
```

The render_to_string shortcut takes one required argument – template_name, which should be the name of the template to load and render (or a list of template names, in which case Django will use the first template in the list that exists) – and two optional arguments:

dictionary

A dictionary to be used as variables and values for the template's context. This can also be passed as the second positional argument.

context_instance

An instance of Context or a subclass (e.g., an instance of RequestContext) to use as the template's context. This can also be passed as the third positional argument.

See also the render_to_response() shortcut, which calls render_to_string and feeds the result into an HttpResponse suitable for returning directly from a view.

Configuring the template system in standalone mode

Note

This section is only of interest to people trying to use the template system as an output component in another application. If you're using the template system as part of a Django application, nothing here applies to you.

Normally, Django will load all the configuration information it needs from its own default configuration file, combined with the settings in the module given in the DJANGO_SETTINGS_MODULE environment variable. But if you're using the template system independently of the rest of Django, the environment variable approach isn't very convenient, because you probably want to configure the template system in line with the rest of your application rather than dealing with settings files and pointing to them via environment variables.

To solve this problem, you need to use the manual configuration option described in Using settings without setting DJANGO_SETTINGS_MODULE. Simply import the appropriate pieces of the templating system and then, before you call any of the templating functions, call django.conf.settings.configure() with any settings you wish to specify. You might want to consider setting at leastTEMPLATE_DIRS (if you're going to use template loaders), DEFAULT_CHARSET (although the default of utf-8 is probably fine) andTEMPLATE_DEBUG. If you plan to use the url template tag, you will also need to set

the ROOT_URLCONF setting. All available settings are described in the settings documentation, and any setting starting with TEMPLATE_ is of obvious interest.

Using an alternative template language

The Django Template and Loader classes implement a simple API for loading and rendering templates. By providing some simple wrapper classes that implement this API we can use third party template systems like Jinja2 or Cheetah. This allows us to use third-party template libraries without giving up useful Django features like the Django Context object and handy shortcuts like render_to_response().

The core component of the Django templating system is the Template class. This class has a very simple interface: it has a constructor that takes a single positional argument specifying the template string, and a render() method that takes a Context object and returns a string containing the rendered response.

Suppose we're using a template language that defines a Template object with a render() method that takes a dictionary rather than a Contextobject. We can write a simple wrapper that implements the Django Template interface:

```
import some_template_language
class Template(some_template_language.Template):
    def render(self, context):
        # flatten the Django Context into a single dictionary.
        context_dict = {}
        for d in context.dicts:
            context_dict.update(d)
        return super(Template, self).render(context_dict)
```

That's all that's required to make our fictional Template class compatible with the Django loading and rendering system!

The next step is to write a Loader class that returns instances of our custom template class instead of the default Template. Custom Loaderclasses should inherit from django.template.loader.BaseLoader and override the load_template_source() method, which takes atemplate_name argument, loads the template from disk (or elsewhere), and returns a tuple: (template_string, template_origin).

The load_template() method of the Loader class retrieves the template string by calling load_template_source(), instantiates a Templatefrom the template source, and returns a tuple: (template, template_origin). Since this is the method that actually instantiates theTemplate, we'll need to override it to use our custom template class instead. We can inherit from the builtindjango.template.loaders.app_directories.Loader to take advantage of the load_template_source() method implemented there:

```
from django.template.loaders import app_directories
class Loader(app_directories.Loader):
    is_usable = True

    def load_template(self, template_name, template_dirs=None):
        source, origin = self.load_template_source(template_name, template_dirs)
        template = Template(source)
        return template, origin
```

Finally, we need to modify our project settings, telling Django to use our custom loader. Now we can write all of our templates in our alternative template language while continuing to use the rest of the Django templating system.

Request and response objects

Quick overview

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an HttpRequest object that contains metadata about the request. Then Django loads the appropriate view, passing the HttpRequest as the first argument to the view function. Each view is responsible for returning an HttpResponse object.

This document explains the APIs for HttpRequest and HttpResponse objects, which are defined in the django.http module.

HttpRequest objects

class HttpRequest

Attributes

All attributes should be considered read-only, unless stated otherwise below. session is a notable exception.

HttpRequest.body

The raw HTTP request body as a byte string. This is useful for processing data in different ways than conventional HTML forms: binary images, XML payload etc. For processing conventional form data, use HttpRequest.POST.

You can also read from an HttpRequest using a file-like interface. See HttpRequest.read().

HttpRequest.path

A string representing the full path to the requested page, not including the domain.

Example: "/music/bands/the_beatles/"

HttpRequest.path_info

Under some Web server configurations, the portion of the URL after the host name is split up into a script prefix portion and a path info portion. The path_info attribute always contains the path info portion of the path, no matter what Web server is being used. Using this instead of pathcan make your code easier to move between test and deployment servers.

For example, if the WSGIScriptAlias for your application is set to "/minfo", then path might be "/minfo/music/bands/the_beatles/"and path_info would be "/music/bands/the_beatles/".

HttpRequest.method

A string representing the HTTP method used in the request. This is guaranteed to be uppercase. Example:

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

HttpRequest.encoding

A string representing the current encoding used to decode form submission data (or None, which means the DEFAULT_CHARSET setting is used). You can write to this attribute to change the encoding used when accessing the form data. Any subsequent attribute accesses (such as reading from GET or POST) will use the new encoding value. Useful if you know the form data is not in the DEFAULT_CHARSET encoding.

HttpRequest.GET

A dictionary-like object containing all given HTTP GET parameters. See the QueryDict documentation below.

HttpRequest.POST

A dictionary-like object containing all given HTTP POST parameters, providing that the request contains form data. See the QueryDictdocumentation below. If you need to access raw or non-form data posted in the request, access this through the HttpRequest.body attribute instead.

Changed in Django 1.5:

Before Django 1.5, HttpRequest.POST contained non-form data.

It’s possible that a request can come in via POST with an empty POST dictionary – if, say, a form is requested via the POST HTTP method but does not include form data. Therefore, you shouldn’t use if request.POST to check for use of the POST method; instead, useif request.method == "POST" (see above).

Note: POST does not include file-upload information. See FILES.

HttpRequest.REQUEST

For convenience, a dictionary-like object that searches POST first, then GET. Inspired by PHP’s $_REQUEST.

For example, if GET = {"name": "john"} and POST = {"age": '34'}, REQUEST["name"] would be "john",

and REQUEST["age"] would be "34".

It's strongly suggested that you use GET and POST instead of REQUEST, because the former are more explicit.

HttpRequest.COOKIES

A standard Python dictionary containing all cookies. Keys and values are strings.

HttpRequest.FILES

A dictionary-like object containing all uploaded files. Each key in FILES is the name from the <input type="file" name="" />. Each value inFILES is an UploadedFile as described below.

See Managing files for more information.

Note that FILES will only contain data if the request method was POST and the <form> that posted to the request hadenctype="multipart/form-data". Otherwise, FILES will be a blank dictionary-like object.

HttpRequest.META

A standard Python dictionary containing all available HTTP headers. Available headers depend on the client and server, but here are some examples:

CONTENT_LENGTH – the length of the request body (as a string).

CONTENT_TYPE – the MIME type of the request body.

HTTP_ACCEPT_ENCODING – Acceptable encodings for the response.

HTTP_ACCEPT_LANGUAGE – Acceptable languages for the response.

HTTP_HOST – The HTTP Host header sent by the client.

HTTP_REFERER – The referring page, if any.

HTTP_USER_AGENT – The client's user-agent string.

QUERY_STRING – The query string, as a single (unparsed) string.

REMOTE_ADDR – The IP address of the client.

REMOTE_HOST – The hostname of the client.

REMOTE_USER – The user authenticated by the Web server, if any.

REQUEST_METHOD – A string such as "GET" or "POST".

SERVER_NAME – The hostname of the server.

SERVER_PORT – The port of the server (as a string).

With the exception of CONTENT_LENGTH and CONTENT_TYPE, as given above, any HTTP headers in the request are converted to META keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an HTTP_ prefix to the name. So, for example, a header called X-Bender would be mapped to the META key HTTP_X_BENDER.

HttpRequest.user

An object of type AUTH_USER_MODEL representing the currently logged-in user. If the user isn't currently logged in, user will be set to an instance of django.contrib.auth.models.AnonymousUser. You can tell them apart with is_authenticated(), like so:

```
if request.user.is_authenticated():
    # Do something for logged-in users.
else:
    # Do something for anonymous users.
```

user is only available if your Django installation has the AuthenticationMiddleware activated. For more, see User authentication in Django.

HttpRequest.session

A readable-and-writable, dictionary-like object that represents the current session. This is only available if your Django installation has session support activated. See the session documentation for full details.

HttpRequest.urlconf

Not defined by Django itself, but will be read if other code (e.g., a custom middleware class) sets it. When present, this will be used as the root URLconf for the current request, overriding the ROOT_URLCONF setting. See How Django processes a request for details.

HttpRequest.resolver_match

New in Django 1.5.

An instance of ResolverMatch representing the resolved url. This attribute is only set after url resolving took place, which means it’s available in all views but not in middleware methods which are executed before url resolving takes place (like process_request, you can useprocess_view instead).

Methods

HttpRequest.get_host()

Returns the originating host of the request using information from the HTTP_X_FORWARDED_HOST (if USE_X_FORWARDED_HOST is enabled) andHTTP_HOST headers, in that order. If they don’t provide a value, the method uses a combination of SERVER_NAME and SERVER_PORT as detailed in PEP 3333.

Example: "127.0.0.1:8000"

Note

The get_host() method fails when the host is behind multiple proxies. One solution is to use middleware to rewrite the proxy headers, as in the following example:

```
class MultipleProxyMiddleware(object):
    FORWARDED_FOR_FIELDS = [
        'HTTP_X_FORWARDED_FOR',
        'HTTP_X_FORWARDED_HOST',
        'HTTP_X_FORWARDED_SERVER',
    ]
    def process_request(self, request):
        """
        Rewrites the proxy headers so that only the most
        recent proxy is used.
        """
        for field in self.FORWARDED_FOR_FIELDS:
            if field in request.META:
                if ',' in request.META[field]:
                    parts = request.META[field].split(',')
                    request.META[field] = parts[-1].strip()
```

This middleware should be positioned before any other middleware that relies on the value of get_host() – for instance,CommonMiddleware or CsrfViewMiddleware.

HttpRequest.get_full_path()

Returns the path, plus an appended query string, if applicable.

Example: "/music/bands/the_beatles/?print=true"

HttpRequest.build_absolute_uri(location)

Returns the absolute URI form of location. If no location is provided, the location will be set to request.get_full_path().

If the location is already an absolute URI, it will not be altered. Otherwise the absolute URI is built using the server variables available in this request.

Example: "http://example.com/music/bands/the_beatles/?print=true"

HttpRequest.get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)

Returns a cookie value for a signed cookie, or raises a django.core.signing.BadSignature exception if the signature is no longer valid. If you provide the default argument the exception will be suppressed and that default value will be returned instead.

The optional salt argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the max_ageargument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than max_age seconds.

For example:

```
>>> request.get_signed_cookie('name')
'Tony'
```

```
>>> request.get_signed_cookie('name', salt='name-salt')
'Tony' # assuming cookie was set using the same salt
>>> request.get_signed_cookie('non-existing-cookie')
...
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name', max_age=60)
...
SignatureExpired: Signature age 1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name', False, max_age=60)
False
```

See cryptographic signing for more information.

HttpRequest.is_secure()

Returns True if the request is secure; that is, if it was made with HTTPS.

HttpRequest.is_ajax()

Returns True if the request was made via an XMLHttpRequest, by checking the HTTP_X_REQUESTED_WITH header for the string'XMLHttpRequest'. Most modern JavaScript libraries send this header. If you write your own XMLHttpRequest call (on the browser side), you'll have to set this header manually if you want is_ajax() to work.

HttpRequest.read(size=None)

HttpRequest.readline()

HttpRequest.readlines()

HttpRequest.xreadlines()

HttpRequest.__iter__()

Methods implementing a file-like interface for reading from an HttpRequest instance. This makes it possible to consume an incoming request in a streaming fashion. A common use-case would be to process a big XML payload with iterative parser without constructing a whole XML tree in memory.

Given this standard interface, an HttpRequest instance can be passed directly to an XML parser such as ElementTree:

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

UploadedFile objects

class UploadedFile

Attributes

UploadedFile.name

The name of the uploaded file.

UploadedFile.size

The size, in bytes, of the uploaded file.

Methods

UploadedFile.chunks(chunk_size=None)

Returns a generator that yields sequential chunks of data.

UploadedFile.read(num_bytes=None)

Read a number of bytes from the file.

QueryDict objects

class QueryDict

In an HttpRequest object, the GET and POST attributes are instances

of django.http.QueryDict. QueryDict is a dictionary-like class customized to deal with multiple values for the same key. This is necessary because some HTML form elements, notably<select multiple="multiple">, pass multiple values for the same key.

QueryDict instances are immutable, unless you create a copy() of them. That means you can't change attributes of request.POST andrequest.GET directly.

Methods

QueryDict implements all the standard dictionary methods, because it's a subclass of dictionary. Exceptions are outlined here:

QueryDict.__getitem__(key)

Returns the value for the given key. If the key has more than one value, __getitem__() returns the last value. Raisesdjango.utils.datastructures.MultiValueDictKeyError if the key does not exist. (This is a subclass of Python's standard KeyError, so you can stick to catching KeyError.)

QueryDict.__setitem__(key, value)

Sets the given key to [value] (a Python list whose single element is value). Note that this, as other dictionary functions that have side effects, can only be called on a mutable QueryDict (one that was created via copy()).

QueryDict.__contains__(key)

Returns True if the given key is set. This lets you do, e.g., if "foo" in request.GET.

QueryDict.get(key, default)

Uses the same logic as __getitem__() above, with a hook for returning a default value if the key doesn't exist.

QueryDict.setdefault(key, default)

Just like the standard dictionary setdefault() method, except it uses __setitem__() internally.

QueryDict.update(other_dict)

Takes either a QueryDict or standard dictionary. Just like the standard dictionary update() method, except it appends to the current dictionary items rather than replacing them. For example:

```
>>> q = QueryDict('a=1')
>>> q = q.copy() # to make it mutable
>>> q.update({'a': '2'})
>>> q.getlist('a')
[u'1', u'2']
>>> q['a'] # returns the last
[u'2']
```

QueryDict.items()

Just like the standard dictionary items() method, except this uses the same last-value logic as __getitem__(). For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[(u'a', u'3')]
```

QueryDict.iteritems()

Just like the standard dictionary iteritems() method. Like QueryDict.items() this uses the same last-value logic asQueryDict.__getitem__().

QueryDict.iterlists()

Like QueryDict.iteritems() except it includes all values, as a list, for each member of the dictionary.

QueryDict.values()

Just like the standard dictionary values() method, except this uses the same last-value logic as __getitem__(). For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
[u'3']
```

QueryDict.itervalues()

Just like QueryDict.values(), except an iterator.

In addition, QueryDict has the following methods:

QueryDict.copy()

Returns a copy of the object, using copy.deepcopy() from the Python standard library. The copy will be mutable – that is, you can change its values.

QueryDict.getlist(key, default)

Returns the data with the requested key, as a Python list. Returns an empty list if the key doesn't exist and no default value was provided. It's guaranteed to return a list of some sort unless the default value was no list.

QueryDict.setlist(key, list_)

Sets the given key to list_ (unlike __setitem__()).

QueryDict.appendlist(key, item)

Appends an item to the internal list associated with key.

QueryDict.setlistdefault(key, default_list)

Just like setdefault, except it takes a list of values instead of a single value.

QueryDict.lists()

Like items(), except it includes all values, as a list, for each member of the dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[(u'a', [u'1', u'2', u'3'])]
```

QueryDict.pop(key)

Returns a list of values for the given key and removes them from the dictionary. Raises KeyError if the key does not exist. For example:

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.pop('a')
[u'1', u'2', u'3']
```

QueryDict.popitem()

Removes an arbitrary member of the dictionary (since there's no concept of ordering), and returns a two value tuple containing the key and a list of all values for the key. Raises KeyError when called on an empty dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.popitem()
(u'a', [u'1', u'2', u'3'])
```

QueryDict.dict()

Returns dict representation of QueryDict. For every (key, list) pair in QueryDict, dict will have (key, item), where item is one element of the list, using same logic as QueryDict.__getitem__():

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
{u'a': u'5'}
```

QueryDict.urlencode([safe])

Returns a string of the data in query-string format. Example:

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

Optionally, urlencode can be passed characters which do not require encoding. For example:

```
>>> q = QueryDict('', mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```

HttpResponse objects

class HttpResponse

In contrast to HttpRequest objects, which are created automatically by Django, HttpResponse objects are

your responsibility. Each view you write is responsible for instantiating, populating and returning an HttpResponse.

The HttpResponse class lives in the django.http module.

Usage

Passing strings

Typical usage is to pass the contents of the page, as a string, to the HttpResponse constructor:

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="text/plain")
```

But if you want to add content incrementally, you can use response as a file-like object:

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

Passing iterators

Finally, you can pass HttpResponse an iterator rather than strings. If you use this technique, the iterator should return strings.

Passing an iterator as content to HttpResponse creates a streaming response if (and only if) no middleware accesses theHttpResponse.content attribute before the response is returned.

Changed in Django 1.5:

This technique is fragile and was deprecated in Django 1.5. If you need the response to be streamed from the iterator to the client, you should use theStreamingHttpResponse class instead.

As of Django 1.7, when HttpResponse is instantiated with an iterator, it will consume it immediately, store the response content as a string, and discard the iterator.

Changed in Django 1.5:

You can now use HttpResponse as a file-like object even if it was instantiated with an iterator. Django will consume and save the content of the iterator on first access.

Setting header fields

To set or remove a header field in your response, treat it like a dictionary:

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

Note that unlike a dictionary, del doesn't raise KeyError if the header field doesn't exist.

For setting the Cache-Control and Vary header fields, it is recommended to use the patch_cache_control() and patch_vary_headers()methods from django.utils.cache, since these fields can have multiple, comma-separated values. The "patch" methods ensure that other values, e.g. added by a middleware, are not removed.

HTTP header fields cannot contain newlines. An attempt to set a header field containing a newline character (CR or LF) will raiseBadHeaderError

Telling the browser to treat the response as a file attachment

To tell the browser to treat the response as a file attachment, use the content_type argument and set the Content-Disposition header. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename="foo.xls"'
```

There's nothing Django-specific about the Content-Disposition header, but it's easy to forget the syntax, so we've included it here.

Attributes

HttpResponse.content

A string representing the content, encoded from a Unicode object if necessary.

HttpResponse.status_code

The HTTP status code for the response.

HttpResponse.reason_phrase

New in Django 1.6.
The HTTP reason phrase for the response.
HttpResponse.streaming
This is always False.
This attribute exists so middleware can treat streaming responses differently from regular responses.
Methods
HttpResponse.__init__(content='', content_type=None, status=200, reason=None)
Instantiates an HttpResponse object with the given page content and content type.
content should be an iterator or a string. If it's an iterator, it should return strings, and those strings will be joined together to form the content of the response. If it is not an iterator or a string, it will be converted to a string when accessed.
content_type is the MIME type optionally completed by a character set encoding and is used to fill the HTTP Content-Type header. If not specified, it is formed by the DEFAULT_CONTENT_TYPE and DEFAULT_CHARSET settings, by default: "text/html; charset=utf-8".
Historically, this parameter was called mimetype (now deprecated).
status is the HTTP status code for the response.
New in Django 1.6.
reason is the HTTP response phrase. If not provided, a default phrase will be used.
HttpResponse.__setitem__(header, value)
Sets the given header name to the given value. Both header and value should be strings.
HttpResponse.__delitem__(header)
Deletes the header with the given name. Fails silently if the header doesn't exist. Case-insensitive.
HttpResponse.__getitem__(header)
Returns the value for the given header name. Case-insensitive.
HttpResponse.has_header(header)
Returns True or False based on a case-insensitive check for a header with the given name.
HttpResponse.set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure =None,httponly=False)
Sets a cookie. The parameters are the same as in the Cookie.Morsel object in the Python standard library.
max_age should be a number of seconds, or None (default) if the cookie should last only as long as the client's browser session. Ifexpires is not specified, it will be calculated.
expires should either be a string in the format "Wdy, DD-Mon-YY HH:MM:SS GMT" or a datetime.datetime object in UTC. If expiresis a datetime object, the max_age will be calculated.
Use domain if you want to set a cross-domain cookie. For example, domain=".lawrence.com" will set a cookie that is readable by the domains www.lawrence.com, blogs.lawrence.com and calendars.lawrence.com. Otherwise, a cookie will only be readable by the domain that set it.
Use httponly=True if you want to prevent client-side JavaScript from having access to the cookie.
HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the RFC 2109 standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data.
HttpResponse.set_signed_cookie(key, value, salt='', max_age=None, expires=None, path='/', domain= None, secure=None,httponly=True)
Like set_cookie(), but cryptographic signing the cookie before setting it. Use in conjunction with HttpRequest.get_signed_cookie(). You can use the optional salt argument for added key strength, but you will need to remember to pass it to the correspondingHttpRequest.get_signed_cookie() call.
HttpResponse.delete_cookie(key, path='/', domain=None)
Deletes the cookie with the given key. Fails silently if the key doesn't exist.
Due to the way cookies work, path and domain should be the same values you used in set_cookie() – otherwise the cookie may not be deleted.
HttpResponse.write(content)
This method makes an HttpResponse instance a file-like object.

HttpResponse.flush()
This method makes an HttpResponse instance a file-like object.
HttpResponse.tell()
This method makes an HttpResponse instance a file-like object.

HttpResponse subclasses

Django includes a number of HttpResponse subclasses that handle different types of HTTP responses. Like HttpResponse, these subclasses live in django.http.
class HttpResponseRedirect
The first argument to the constructor is required – the path to redirect to. This can be a fully qualified URL (e.g.'http://www.yahoo.com/search/') or an absolute path with no domain (e.g. '/search/'). See HttpResponse for other optional constructor arguments. Note that this returns an HTTP status code 302.
url

New in Django 1.6.

This read-only attribute represents the URL the response will redirect to (equivalent to the Location response header).
class HttpResponsePermanentRedirect
Like HttpResponseRedirect, but it returns a permanent redirect (HTTP status code 301) instead of a "found" redirect (status code 302).
class HttpResponseNotModified
The constructor doesn't take any arguments and no content should be added to this response. Use this to designate that a page hasn't been modified since the user's last request (status code 304).
class HttpResponseBadRequest
Acts just like HttpResponse but uses a 400 status code.
class HttpResponseNotFound
Acts just like HttpResponse but uses a 404 status code.
class HttpResponseForbidden
Acts just like HttpResponse but uses a 403 status code.
class HttpResponseNotAllowed
Like HttpResponse, but uses a 405 status code. The first argument to the constructor is required: a list of permitted methods (e.g.['GET', 'POST']).
class HttpResponseGone
Acts just like HttpResponse but uses a 410 status code.
class HttpResponseServerError
Acts just like HttpResponse but uses a 500 status code.

Note

If a custom subclass of HttpResponse implements a render method, Django will treat it as emulating aSimpleTemplateResponse, and the render method must itself return a valid response object.
StreamingHttpResponse objects
New in Django 1.5.
class StreamingHttpResponse
The StreamingHttpResponse class is used to stream a response from Django to the browser. You might want to do this if generating the response takes too long or uses too much memory. For instance, it's useful for generating large CSV files.

Performance considerations

Django is designed for short-lived requests. Streaming responses will tie a worker process for the entire duration of the response. This may result in poor performance.
Generally speaking, you should perform expensive tasks outside of the request-response cycle, rather than resorting to a streamed response.
The StreamingHttpResponse is not a subclass of HttpResponse, because it features a slightly different API. However, it is almost identical, with the following notable differences:

It should be given an iterator that yields strings as content.
You cannot access its content, except by iterating the response object itself. This should only occur when the response is returned to the client.
It has no content attribute. Instead, it has a streaming_content attribute.
You cannot use the file-like object tell() or write() methods. Doing so will raise an exception.
StreamingHttpResponse should only be used in situations where it is absolutely required that the whole content isn't iterated before transferring the data to the client. Because the content can't be accessed, many middlewares can't function normally. For example the ETag andContent- Length headers can't be generated for streaming responses.
Attributes
StreamingHttpResponse.streaming_content
An iterator of strings representing the content.
HttpResponse.status_code
The HTTP status code for the response.
HttpResponse.reason_phrase
New in Django 1.6.
The HTTP reason phrase for the response.
HttpResponse.streaming
This is always True.

Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for simple tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the class-based views reference documentation.
Introduction to Class-based views
Class-based generic views
Form handling with class-based views
Using mixins with class-based views

Basic examples

Django provides base view classes which will suit a wide range of applications. All views inherit from the View class, which handles linking the view in to the URLs, HTTP method dispatching and other simple features. RedirectView is for a simple HTTP redirect, and TemplateViewextends the base class to make it also render a template.

Simple usage in your URLconf

The simplest way to use generic views is to create them directly in your URLconf. If you're only changing a few simple attributes on a class-based view, you can simply pass them into the as_view() method call itself:

```
from django.conf.urls import patterns
from django.views.generic import TemplateView
urlpatterns = patterns('',
    (r'^about/', TemplateView.as_view(template_name="about.html")),
)
```

Any arguments passed to as_view() will override attributes set on the class. In this example, we set template_name on the TemplateView. A similar overriding pattern can be used for the url attribute on RedirectView.

Subclassing generic views

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the template_name) or methods (such as get_context_data) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, about.html. Django

has a generic view to do this - TemplateView - so we can just subclass it, and override the template name:

```
# some_app/views.py
from django.views.generic import TemplateView
class AboutView(TemplateView):
    template_name = "about.html"
```

Then we just need to add this new view into our URLconf. TemplateView is a class, not a function, so we point the URL to the as_view() class method instead, which provides a function-like entry to class-based views:

```
# urls.py
from django.conf.urls import patterns
from some_app.views import AboutView

urlpatterns = patterns('',
    (r'^about/', AboutView.as_view()),
)
```

For more information on how to use the built in generic views, consult the next topic on generic class based views.

Supporting other HTTP methods

Suppose somebody wants to access our book library over HTTP using the views as an API. The API client would connect every now and then and download book data for the books published since last visit. But if no new books appeared since then, it is a waste of CPU time and bandwidth to fetch the books from the database, render a full response and send it to the client. It might be preferable to ask the API when the most recent book was published.

We map the URL to book list view in the URLconf:

```
from django.conf.urls import patterns
from books.views import BookListView

urlpatterns = patterns('',
    (r'^books/$', BookListView.as_view()),
)
```

And the view:

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book

class BookListView(ListView):
    model = Book

    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest('publication_date')
        response = HttpResponse('')
        # RFC 1123 date format
        response['Last-Modified'] = last_book.publication_date.strftime('%a, %d %b %Y %H:%M:%S GMT')
        return response
```

If the view is accessed from a GET request, a plain-and-simple object list is returned in the response (using book_list.html template). But if the client issues a HEAD request, the response has an empty body and the Last-Modified header indicates when the most recent book was published. Based on this information, the client may or may not download the full object list.

Class-based views

Class-based views API reference. For introductory material, see Class-based views.

Base views View TemplateView RedirectView Generic display views DetailView ListView Generic editing

views FormView CreateView UpdateView DeleteView Generic date views ArchiveIndexView YearArchiveView MonthArchiveView WeekArchiveView DayArchiveView TodayArchiveView DateDetailView Class-based views mixins Simple mixins ContextMixin TemplateResponseMixin Single object mixins SingleObjectMixin SingleObjectTemplateResponseMixin Multiple object mixins MultipleObjectMixin MultipleObjectTemplateResponseMixin Editing mixins FormMixin ModelFormMixin ProcessFormView Date-based mixins YearMixin MonthMixin DayMixin WeekMixin DateMixin BaseDateListView Class-based generic views - flattened index Simple generic views View TemplateView RedirectView Detail Views DetailView List Views ListView Editing views FormView CreateView UpdateView DeleteView Date-based views ArchiveIndexView YearArchiveView MonthArchiveView WeekArchiveView DayArchiveView TodayArchiveView DateDetailView

Specification

Each request served by a class-based view has an independent state; therefore, it is safe to store state variables on the instance (i.e.,self.foo = 3 is a thread-safe operation).

A class-based view is deployed into a URL pattern using the as_view() classmethod:

```
urlpatterns = patterns('',
    (r'^view/$', MyView.as_view(size=42)),
)
```

Thread safety with view arguments

Arguments passed to a view are shared between every instance of a view. This means that you shouldn't use a list, dictionary, or any other mutable object as an argument to a view. If you do and the shared object is modified, the actions of one user visiting your view could have an effect on subsequent users visiting the same view.

Arguments passed into as_view() will be assigned onto the instance that is used to service a request. Using the previous example, this means that every request on MyView is able to use self.size. Arguments must correspond to attributes that already exist on the class (return True on ahasattr check).

Base vs Generic views

Base class-based views can be thought of as parent views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

Django's generic views are built off of those base views, and were developed as a shortcut for common usage patterns such as displaying the details of an object. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.

Most generic views require the queryset key, which is a QuerySet instance; see Making queries for more information about QuerySet objects.

Introduction to Class-based views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

Organization of code related to specific HTTP methods (GET, POST, etc) can be addressed by separate methods instead of conditional branching.

Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

The relationship and history of generic views, class-based views, and class-based generic views

In the beginning there was only the view function contract, Django passed your function an HttpRequest and expected back an HttpResponse. This was the extent of what Django provided.

Early on it was recognized that there were common idioms and patterns found in view development. Function-based generic views were introduced to abstract these patterns and ease view development for the common cases.

The problem with function-based generic views is that while they covered the simple cases well, there was

no way to extend or customize them beyond some simple configuration options, limiting their usefulness in many real-world applications.

Class-based generic views were created with the same objective as function-based generic views, to make view development easier. However, the way the solution is implemented, through the use of mixins, provides a toolkit that results in class-based generic views being more extensible and flexible than their function-based counterparts.

If you have tried function based generic views in the past and found them lacking, you should not think of class-based generic views as simply a class-based equivalent, but rather as a fresh approach to solving the original problems that generic views were meant to solve.

The toolkit of base classes and mixins that Django uses to build class-based generic views are built for maximum flexibility, and as such have many hooks in the form of default method implementations and attributes that you are unlikely to be concerned with in the simplest use cases. For example, instead of limiting you to a class based attribute for form_class, the implementation uses a get_form method, which calls aget_form_class method, which in its default implementation just returns the form_class attribute of the class. This gives you several options for specifying what form to use, from a simple attribute, to a fully dynamic, callable hook. These options seem to add hollow complexity for simple situations, but without them, more advanced designs would be limited.

Using class-based views

At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So where the code to handle HTTP GET in a view function would look something like:

```
from django.http import HttpResponse
def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponse('result')
```

In a class-based view, this would become:

```
from django.http import HttpResponse
from django.views.generic.base import View
class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

Because Django's URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an as_view() class method which serves as the callable entry point to your class. The as_view entry point creates an instance of your class and calls its dispatch() method. dispatch looks at the request to determine whether it is a GET, POST, etc, and relays the request to a matching method if one is defined, or raises HttpResponseNotAllowed if not:

```
# urls.py
from django.conf.urls import patterns
from myapp.views import MyView

urlpatterns = patterns('',
    (r'^about/', MyView.as_view()),
)
```

It is worth noting that what your method returns is identical to what you return from a function-based view, namely some form of HttpResponse. This means that http shortcuts or TemplateResponse objects are valid to use inside a class-based view.

While a minimal class-based view does not require any class attributes to perform its job, class attributes are useful in many class-based designs, and there are two ways to configure or set class attributes.

The first is the standard Python way of subclassing and overriding attributes and methods in the subclass. So

that if your parent class had an attribute greeting like this:

```
from django.http import HttpResponse
from django.views.generic.base import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

You can override that in a subclass:

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Another option is to configure class attributes as keyword arguments to the as_view() call in the URLconf:

```
urlpatterns = patterns('',
    (r'^about/', GreetingView.as_view(greeting="G'day")),
)
```

Note

While your class is instantiated for each request dispatched to it, class attributes set through the as_view() entry point are configured only once at the time your URLs are imported.

Using mixins

Mixins are a form of multiple inheritance where behaviors and attributes of multiple parent classes can be combined.

For example, in the generic class-based views there is a mixin called TemplateResponseMixin whose primary purpose is to define the methodrender_to_response(). When combined with the behavior of the View base class, the result is a TemplateView class that will dispatch requests to the appropriate matching methods (a behavior defined in the View base class), and that has a render_to_response() method that uses a template_name attribute to return a TemplateResponse object (a behavior defined in the TemplateResponseMixin).

Mixins are an excellent way of reusing code across multiple classes, but they come with some cost. The more your code is scattered among mixins, the harder it will be to read a child class and know what exactly it is doing, and the harder it will be to know which methods from which mixins to override if you are subclassing something that has a deep inheritance tree.

Note also that you can only inherit from one generic view - that is, only one parent class may inherit from View and the rest (if any) should be mixins. Trying to inherit from more than one class that inherits from View - for example, trying to use a form at the top of a list and combiningProcessFormView and ListView - won’t work as expected.

Handling forms with class-based views

A basic function-based view that handles forms may look something like this:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import MyForm
def myview(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')
    else:
        form = MyForm(initial={'key': 'value'})

    return render(request, 'form_template.html', {'form': form})
```

A similar class-based view might look like:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views.generic.base import View

from .forms import MyForm

class MyFormView(View):
    form_class = MyForm
    initial = {'key': 'value'}
    template_name = 'form_template.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')

        return render(request, self.template_name, {'form': form})
```

This is a very simple case, but you can see that you would then have the option of customizing this view by overriding any of the class attributes, e.g. form_class, via URLconf configuration, or subclassing and overriding one or more of the methods (or both!).

Decorating class-based views

The extension of class-based views isn't limited to using mixins. You can use also use decorators. Since class-based views aren't functions, decorating them works differently depending on if you're using as_view or creating a subclass.

Decorating in URLconf

The simplest way of decorating class-based views is to decorate the result of the as_view() method. The easiest place to do this is in the URLconf where you deploy your view:

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = patterns('',
    (r'^about/', login_required(TemplateView.as_view(template_name="secret.html"))),
    (r'^vote/', permission_required('polls.can_vote')(VoteView.as_view())),
)
```

This approach applies the decorator on a per-instance basis. If you want every instance of a view to be decorated, you need to take a different approach.

Decorating the class

To decorate every instance of a class-based view, you need to decorate the class definition itself. To do this you apply the decorator to thedispatch() method of the class.

A method on a class isn't quite the same as a standalone function, so you can't just apply a function decorator to the method – you need to transform it into a method decorator first. The method_decorator decorator transforms a function decorator into a method decorator so that it can be

used on an instance method. For example:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

class ProtectedView(TemplateView):
    template_name = 'secret.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(ProtectedView, self).dispatch(*args, **kwargs)
```

In this example, every instance of ProtectedView will have login protection.

Note

method_decorator passes *args and **kwargs as parameters to the decorated method on the class. If your method does not accept a compatible set of parameters it will raise a TypeError exception.

Class-based generic views

Writing Web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but Web developers also experience this boredom at the view level.

Django's generic views were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

We can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of any object. Then the model in question can be passed as an extra argument to the URLconf.

Django ships with generic views to do the following:

Display list and detail pages for a single object. If we were creating an application to manage conferences then a TalkListView and aRegisteredUserListView would be examples of list views. A single talk page is an example of what we call a "detail" view.

Present date-based objects in year/month/day archive pages, associated detail, and "latest" pages.

Allow users to create, update, and delete objects – with or without authorization.

Taken together, these views provide easy interfaces to perform the most common tasks developers encounter.

Extending generic views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

This is one of the reasons generic views were redesigned for the 1.3 release - previously, they were just view functions with a bewildering array of options; now, rather than passing in a large amount of configuration in the URLconf, the recommended way to extend generic views is to subclass them, and override their attributes or methods.

That said, generic views will have a limit. If you find you're struggling to implement your view as a subclass of a generic view, then you may find it more effective to write just the code you need, using your own class-based or functional views.

More examples of generic views are available in some third party applications, or you could write your own as needed.

Generic views of objects

TemplateView certainly is useful, but Django's generic views really shine when it comes to presenting views of your database content. Because it's such a common task, Django comes with a handful of built-in generic views that make generating list and detail views of objects incredibly easy.

Let’s start by looking at some examples of showing a list of objects or an individual object. We’ll be using these models:

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    # On Python 3: def __str__(self):
    def __unicode__(self):
        return self.name

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    # On Python 3: def __str__(self):
    def __unicode__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

Now we need to define a view:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

Finally hook that view into your urls:

```
# urls.py
from django.conf.urls import patterns, url
from books.views import PublisherList

urlpatterns = patterns('',
    url(r'^publishers/$', PublisherList.as_view()),
)
```

That’s all the Python code we need to write. We still need to write a template, however. We could explicitly tell the view which template to use by adding a template_name attribute to the view, but in the absence of an

explicit template Django will infer one from the object's name. In this case, the inferred template will be "books/publisher_list.html" – the "books" part comes from the name of the app that defines the model, while the "publisher" bit is just the lowercased version of the model's name.

Note

Thus, when (for example) the django.template.loaders.app_directories.Loader template loader is enabled inTEMPLATE_LOADERS, a template location could be: /path/to/project/books/templates/books/publisher_list.html

This template will be rendered against a context containing a variable called object_list that contains all the publisher objects. A very simple template might look like the following:

```
{% extends "base.html" %}

{% block content %}
  <h2>Publishers</h2>
  <ul>
    {% for publisher in object_list %}
      <li>{{ publisher.name }}</li>
    {% endfor %}
  </ul>
{% endblock %}
```

That's really all there is to it. All the cool features of generic views come from changing the attributes set on the generic view. The generic views reference documents all the generic views and their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

Making "friendly" template contexts

You might have noticed that our sample publisher list template stores all the publishers in a variable named object_list. While this works just fine, it isn't all that "friendly" to template authors: they have to "just know" that they're dealing with publishers here.

Well, if you're dealing with a model object, this is already done for you. When you are dealing with an object or queryset, Django is able to populate the context using the lower cased version of the model class' name. This is provided in addition to the default object_list entry, but contains exactly the same data, i.e. publisher_list.

If this still isn't a good match, you can manually set the name of the context variable. The context_object_name attribute on a generic view specifies the context variable to use:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
  model = Publisher
  context_object_name = 'my_favourite_publishers'
```

Providing a useful context_object_name is always a good idea. Your coworkers who design templates will thank you.

Adding extra context

Often you simply need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the books on each publisher detail page. The DetailView generic view provides the publisher to the context, but how do we get additional information in that template?

The answer is to subclass DetailView and provide your own implementation of the get_context_data method. The default implementation simply adds the object being displayed to the template, but you can override it to send more:

```
from django.views.generic import DetailView
from books.models import Publisher, Book
```

```
class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

Note

Generally, get_context_data will merge the context data of all parent classes with those of the current class. To preserve this behavior in your own classes where you want to alter the context, you should be sure to call get_context_data on the super class. When no two classes try to define the same key, this will give the expected results. However if any class attempts to override a key after parent classes have set it (after the call to super), any children of that class will also need to explicitly set it after super if they want to be sure to override all parents. If you're having trouble, review the method resolution order of your view.

Viewing subsets of objects

Now let's take a closer look at the model argument we've been using all along. The model argument, which specifies the database model that the view will operate upon, is available on all the generic views that operate on a single object or a collection of objects. However, the model argument is not the only way to specify the objects that the view will operate upon – you can also specify the list of objects using the queryset argument:

```
from django.views.generic import DetailView
from books.models import Publisher

class PublisherDetail(DetailView):

    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

Specifying model = Publisher is really just shorthand for saying queryset = Publisher.objects.all(). However, by using queryset to define a filtered list of objects you can be more specific about the objects that will be visible in the view (see Making queries for more information about QuerySet objects, and see the class-based views reference for the complete details).

To pick a simple example, we might want to order a list of books by publication date, with the most recent first:

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

That's a pretty simple example, but it illustrates the idea nicely. Of course, you'll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):
```

```
    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='Acme Publishing')
    template_name = 'books/acme_list.html'
```

Notice that along with a filtered queryset, we're also using a custom template name. If we didn't, the generic view would use the same template as the "vanilla" object list, which might not be what we want.

Also notice that this isn't a very elegant way of doing publisher-specific books. If we want to add another publisher page, we'd need another handful of lines in the URLconf, and more than a few publishers would get unreasonable. We'll deal with this problem in the next section.

Note

If you get a 404 when requesting /books/acme/, check to ensure you actually have a Publisher with the name 'ACME Publishing'. Generic views have an allow_empty parameter for this case. See the class-based-views reference for more details.

Dynamic filtering

Another common need is to filter down the objects given in a list page by some key in the URL. Earlier we hard-coded the publisher's name in the URLconf, but what if we wanted to write a view that displayed all the books by some arbitrary publisher?

Handily, the ListView has a get_queryset() method we can override. Previously, it has just been returning the value of the queryset attribute, but now we can add more logic.

The key part to making this work is that when class-based views are called, various useful things are stored on self; as well as the request (self.request) this includes the positional (self.args) and name-based (self.kwargs) arguments captured according to the URLconf.

Here, we have a URLconf with a single captured group:

```
# urls.py
from django.conf.urls import patterns
from books.views import PublisherBookList

urlpatterns = patterns('',
    (r'^books/([\w-]+)/$', PublisherBookList.as_view()),
)
```

Next, we'll write the PublisherBookList view itself:

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

As you can see, it's quite easy to add more logic to the queryset selection; if we wanted, we could use self.request.user to filter using the current user, or other more complex logic.

We can also add the publisher into the context at the same time, so we can use it in the template:

```
# ...

def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super(PublisherBookList, self).get_context_data(**kwargs)
    # Add in the publisher
```

```
    context['publisher'] = self.publisher
    return context
```

Performing extra work

The last common pattern we'll look at involves doing some extra work before or after calling the generic view.

Imagine we had a last_accessed field on our Author model that we were using to keep track of the last time anybody looked at that author:

```
# models.py
from django.db import models

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')
    last_accessed = models.DateTimeField()
```

The generic DetailView class, of course, wouldn't know anything about this field, but once again we could easily write a custom view to keep that field updated.

First, we'd need to add an author detail bit in the URLconf to point to a custom view:

```
from django.conf.urls import patterns, url
from books.views import AuthorDetailView

urlpatterns = patterns('',
    #...
    url(r'^authors/(?P<pk>\d+)/$', AuthorDetailView.as_view(), name='author-detail'),
)
```

Then we'd write our new view – get_object is the method that retrieves the object – so we simply override it and wrap the call:

```
from django.views.generic import DetailView
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):

    queryset = Author.objects.all()

    def get_object(self):
        # Call the superclass
        object = super(AuthorDetailView, self).get_object()
        # Record the last accessed date
        object.last_accessed = timezone.now()
        object.save()
        # Return the object
        return object
```

Note

The URLconf here uses the named group pk - this name is the default name that DetailView uses to find the value of the primary key used to filter the queryset.

If you want to call the group something else, you can set pk_url_kwarg on the view. More details can be found in the reference forDetailView

Form handling with class-based views

Form processing generally has 3 paths:
Initial GET (blank or prepopulated form)
POST with invalid data (typically redisplay form with errors)
POST with valid data (process the data and typically redirect)
Implementing this yourself often results in a lot of repeated boilerplate code (see Using a form in a view). To help avoid this, Django provides a collection of generic class-based views for form processing.
Basic Forms
Given a simple contact form:

```
# forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

The view can be constructed using a FormView:

```
# views.py
from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

Notes:
FormView inherits TemplateResponseMixin so template_name can be used here.
The default implementation for form_valid() simply redirects to the success_url.

Model Forms

Generic views really shine when working with models. These generic views will automatically create a ModelForm, so long as they can work out which model class to use:
If the model attribute is given, that model class will be used.
If get_object() returns an object, the class of that object will be used.
If a queryset is given, the model for that queryset will be used.
Model form views provide a form_valid() implementation that saves the model automatically. You can override this if you have any special requirements; see below for examples.
You don’t even need to provide a success_url for CreateView or UpdateView - they will use get_absolute_url() on the model object if available.
If you want to use a custom ModelForm (for instance to add extra validation) simply set form_class on your view.
Note
When specifying a custom form class, you must still specify the model, even though the form_class may be

a ModelForm.
First we need to add get_absolute_url() to our Author class:

```
# models.py
from django.core.urlresolvers import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

Then we can use CreateView and friends to do the actual work. Notice how we're just configuring the generic class-based views here; we don't have to write any logic ourselves:

```
# views.py
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

Note
We have to use reverse_lazy() here, not just reverse as the urls are not loaded when the file is imported.
Changed in Django 1.6.
In Django 1.6, the fields attribute was added, which works the same way as the fields attribute on the inner Meta class on ModelForm.
Omitting the fields attribute will work as previously, but is deprecated and this attribute will be required from 1.8 (unless you define the form class in another way).
Finally, we hook these new views into the URLconf:

```
# urls.py
from django.conf.urls import patterns, url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = patterns('',
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author_add'),
    url(r'author/(?P<pk>\d+)/$', AuthorUpdate.as_view(), name='author_update'),
    url(r'author/(?P<pk>\d+)/delete/$', AuthorDelete.as_view(), name='author_delete'),
)
```

Note
These views inherit SingleObjectTemplateResponseMixin which uses template_name_suffix to construct thetemplate_name based on the model.
In this example:
CreateView and UpdateView use myapp/author_form.html

DeleteView uses myapp/author_confirm_delete.html
If you wish to have separate templates for CreateView and UpdateView, you can set either template_name ortemplate_name_suffix on your view class.

Models and request.user

To track the user that created an object using a CreateView, you can use a custom ModelForm to do this. First, add the foreign key relation to the model:

```
# models.py
from django.contrib.auth import User
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User)

    # ...
```

In the view, ensure that you don't include created_by in the list of fields to edit, and override form_valid() to add the user:

```
# views.py
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

Note that you'll need to decorate this view using login_required(), or alternatively handle unauthorized users in the form_valid().

AJAX example

Here is a simple example showing how you might go about implementing a form that works for AJAX requests as well as 'normal' form POSTs:

```
import json

from django.http import HttpResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    """
    Mixin to add AJAX support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """
    def render_to_json_response(self, context, **response_kwargs):
        data = json.dumps(context)
        response_kwargs['content_type'] = 'application/json'
        return HttpResponse(data, **response_kwargs)

    def form_invalid(self, form):
        response = super(AjaxableResponseMixin, self).form_invalid(form)
```

```
        if self.request.is_ajax():
            return self.render_to_json_response(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super(AjaxableResponseMixin, self).form_valid(form)
        if self.request.is_ajax():
            data = {
                'pk': self.object.pk,
            }
            return self.render_to_json_response(data)
        else:
            return response

class AuthorCreate(AjaxableResponseMixin, CreateView):
    model = Author
    fields = ['name']
```

Using mixins with class-based views

Caution

This is an advanced topic. A working knowledge of Django's class-based views is advised before exploring these techniques.

Django's built-in class-based views provide a lot of functionality, but some of it you may want to use separately. For instance, you may want to write a view that renders a template to make the HTTP response, but you can't use TemplateView; perhaps you need to render a template only on POST, with GET doing something else entirely. While you could use TemplateResponse directly, this will likely result in duplicate code.

For this reason, Django also provides a number of mixins that provide more discrete functionality. Template rendering, for instance, is encapsulated in the TemplateResponseMixin. The Django reference documentation contains full documentation of all the mixins.

Context and template responses

Two central mixins are provided that help in providing a consistent interface to working with templates in class-based views.

TemplateResponseMixin

Every built in view which returns a TemplateResponse will call the render_to_response() method that TemplateResponseMixin provides. Most of the time this will be called for you (for instance, it is called by the get() method implemented by both TemplateView and DetailView); similarly, it's unlikely that you'll need to override it, although if you want your response to return something not rendered via a Django template then you'll want to do it. For an example of this, see the JSONResponseMixin example.

render_to_response() itself calls get_template_names(), which by default will just look up template_name on the class-based view; two other mixins (SingleObjectTemplateResponseMixin and MultipleObjectTemplateResponseMixin) override this to provide more flexible defaults when dealing with actual objects.

New in Django 1.5.

ContextMixin

Every built in view which needs context data, such as for rendering a template

(including TemplateResponseMixin above), should callget_context_data() passing any data they want to ensure is in there as keyword arguments. get_context_data() returns a dictionary; inContextMixin it simply returns its keyword arguments, but it is common to override this to add more members to the dictionary.

Building up Django's generic class-based views

Let's look at how two of Django's generic class-based views are built out of mixins providing discrete functionality. We'll consider DetailView, which renders a "detail" view of an object, and ListView, which will render a list of objects, typically from a queryset, and optionally paginate them. This will introduce us to four mixins which between them provide useful functionality when working with either a single Django object, or multiple objects.

There are also mixins involved in the generic edit views (FormView, and the model-specific views CreateView, UpdateView and DeleteView), and in the date-based generic views. These are covered in the mixin reference documentation.

DetailView: working with a single Django object

To show the detail of an object, we basically need to do two things: we need to look up the object and then we need to make aTemplateResponse with a suitable template, and that object as context.

To get the object, DetailView relies on SingleObjectMixin, which provides a get_object() method that figures out the object based on the URL of the request (it looks for pk and slug keyword arguments as declared in the URLConf, and looks the object up either from the modelattribute on the view, or the queryset attribute if that's provided). SingleObjectMixin also overrides get_context_data(), which is used across all Django's built in class-based views to supply context data for template renders.

To then make a TemplateResponse, DetailView uses SingleObjectTemplateResponseMixin, which extends TemplateResponseMixin, overriding get_template_names() as discussed above. It actually provides a fairly sophisticated set of options, but the main one that most people are going to use is <app_label>/<object_name>_detail.html. The _detail part can be changed by settingtemplate_name_suffix on a subclass to something else. (For instance, the generic edit views use _form for create and update views, and_confirm_delete for delete views.)

ListView: working with many Django objects

Lists of objects follow roughly the same pattern: we need a (possibly paginated) list of objects, typically a QuerySet, and then we need to make aTemplateResponse with a suitable template using that list of objects.

To get the objects, ListView uses MultipleObjectMixin, which provides both get_queryset() and paginate_queryset(). Unlike withSingleObjectMixin, there's no need to key off parts of the URL to figure out the queryset to work with, so the default just uses the queryset ormodel attribute on the view class. A common reason to override get_queryset() here would be to dynamically vary the objects, such as depending on the current user or to exclude posts in the future for a blog.

MultipleObjectMixin also overrides get_context_data() to include appropriate context variables for pagination (providing dummies if pagination is disabled). It relies on object_list being passed in as a keyword argument, which ListView arranges for it.

To make a TemplateResponse, ListView then uses MultipleObjectTemplateResponseMixin; as withSingleObjectTemplateResponseMixin above, this overrides get_template_names() to provide a range of options, with the most commonly-used being <app_label>/<object_name>_list.html, with the _list part again being taken from the template_name_suffixattribute. (The date based generic views use suffixes such as _archive, _archive_year and so on to use different templates for the various specialised date-based list views.)

Using Django's class-based view mixins

Now we've seen how Django's generic class-based views use the provided mixins, let's look at other ways we can combine them. Of course we're still going to be combining them with either built-in class-based views, or other generic class-based views, but there are a range of rarer problems you can solve than are provided for by Django out of the box.

Warning

Not all mixins can be used together, and not all generic class based views can be used with all other mixins. Here we present a few examples that do work; if you want to bring together other functionality then you'll have to consider interactions between attributes and methods that overlap between the different classes you're using, and how method resolution order will affect which versions of the methods will be called in what order.

The reference documentation for Django's class-based views and class-based view mixins will help you in understanding which attributes and methods are likely to cause conflict between different classes and mixins.

If in doubt, it's often better to back off and base your work on View or TemplateView, perhaps with SingleObjectMixin andMultipleObjectMixin. Although you will probably end up writing more code, it is more likely to be clearly understandable to someone else coming to it later, and with fewer interactions to worry about you will save yourself some thinking. (Of course, you can always dip into Django's implementation of the generic class based views for inspiration on how to tackle problems.)

Using SingleObjectMixin with View

If we want to write a simple class-based view that responds only to POST, we'll subclass View and write a post() method in the subclass. However if we want our processing to work on a particular object, identified from the URL, we'll want the functionality provided bySingleObjectMixin.

We'll demonstrate this with the publisher modelling we used in the generic class-based views introduction.

```
# views.py
from django.http import HttpResponseForbidden, HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.views.generic import View
from django.views.generic.detail import SingleObjectMixin
from books.models import Author

class RecordInterest(SingleObjectMixin, View):
    """Records the current user's interest in an author."""
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(reverse('author-detail', kwargs={'pk': self.object.pk}))
```

In practice you'd probably want to record the interest in a key-value store rather than in a relational database, so we've left that bit out. The only bit of the view that needs to worry about using SingleObjectMixin is where we want to look up the author we're interested in, which it just does with a simple call to self.get_object(). Everything else is taken care of for us by the mixin.

We can hook this into our URLs easily enough:

```
# urls.py
from django.conf.urls import patterns, url
from books.views import RecordInterest

urlpatterns = patterns('',
    #...
    url(r'^author/(?P<pk>\d+)/interest/$', RecordInterest.as_view(), name='author-interest'),
```

)

Note the pk named group, which get_object() uses to look up the Author instance. You could also use a slug, or any of the other features ofSingleObjectMixin.

Using SingleObjectMixin with ListView

ListView provides built-in pagination, but you might want to paginate a list of objects that are all linked (by a foreign key) to another object. In our publishing example, you might want to paginate through all the books by a particular publisher.

One way to do this is to combine ListView with SingleObjectMixin, so that the queryset for the paginated list of books can hang off the publisher found as the single object. In order to do this, we need to have two different querysets:

Book queryset for use by ListView

Since we have access to the Publisher whose books we want to list, we simply override get_queryset() and use the Publisher's reverse foreign key manager.

Publisher queryset for use in get_object()

We'll rely on the default implementation of get_object() to fetch the correct Publisher object. However, we need to explicitly pass aqueryset argument because otherwise the default implementation of get_object() would call get_queryset() which we have overridden to return Book objects instead of Publisher ones.

Note

We have to think carefully about get_context_data(). Since both SingleObjectMixin and ListView will put things in the context data under the value of context_object_name if it's set, we'll instead explictly ensure the Publisher is in the context data.ListView will add in the suitable page_obj and paginator for us providing we remember to call super().

Now we can write a new PublisherDetail:

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher

class PublisherDetail(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Publisher.objects.all())
        return super(PublisherDetail, self).get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        context['publisher'] = self.object
        return context

    def get_queryset(self):
        return self.object.book_set.all()
```

Notice how we set self.object within get() so we can use it again later in get_context_data() and get_queryset(). If you don't settemplate_name, the template will default to the normal ListView choice, which in this case would be "books/book_list.html" because it's a list of books; ListView knows nothing about SingleObjectMixin, so it doesn't have any clue this view is anything to do with a Publisher.

The paginate_by is deliberately small in the example so you don't have to create lots of books to see the pagination working! Here's the template you'd want to use:

```
{% extends "base.html" %}
```

```
{% block content %}
  <h2>Publisher {{ publisher.name }}</h2>

  <ol>
   {% for book in page_obj %}
    <li>{{ book.title }}</li>
   {% endfor %}
  </ol>

  <div class="pagination">
    <span class="step-links">
      {% if page_obj.has_previous %}
        <a href="?page={{ page_obj.previous_page_number }}">previous</a>
      {% endif %}

      <span class="current">
        Page {{ page_obj.number }} of {{ paginator.num_pages }}.
      </span>

      {% if page_obj.has_next %}
        <a href="?page={{ page_obj.next_page_number }}">next</a>
      {% endif %}
    </span>
  </div>
{% endblock %}
```

Avoid anything more complex

Generally you can use TemplateResponseMixin and SingleObjectMixin when you need their functionality. As shown above, with a bit of care you can even combine SingleObjectMixin with ListView. However things get increasingly complex as you try to do so, and a good rule of thumb is:

Hint

Each of your views should use only mixins or views from one of the groups of generic class-based views: detail, list, editing and date. For example it's fine to combine TemplateView (built in view) with MultipleObjectMixin (generic list), but you're likely to have problems combining SingleObjectMixin (generic detail) with MultipleObjectMixin (generic list).

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. First, let's look at a naive attempt to combine DetailView with FormMixin to enable use to POST a Django Form to the same URL as we're displaying an object using DetailView.

Using FormMixin with DetailView

Think back to our earlier example of using View and SingleObjectMixin together. We were recording a user's interest in a particular author; say now that we want to let them leave a message saying why they like them. Again, let's assume we're not going to store this in a relational database but instead in something more esoteric that we won't worry about here.

At this point it's natural to reach for a Form to encapsulate the information sent from the user's browser to Django. Say also that we're heavily invested in REST, so we want to use the same URL for displaying the author as for capturing the message from the user. Let's rewrite ourAuthorDetailView to do that.

We'll keep the GET handling from DetailView, although we'll have to add a Form into the context data so we can render it in the template. We'll also want to pull in form processing from FormMixin, and write a bit of code so that on POST the form gets called appropriately.

Note

We use FormMixin and implement post() ourselves rather than try to mix DetailView with FormView (which provides a suitablepost() already) because both of the views implement get(), and things would get much more confusing.

Our new AuthorDetail looks like this:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseForbidden
from django.core.urlresolvers import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDetail(FormMixin, DetailView):
    model = Author
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.object.pk})

    def get_context_data(self, **kwargs):
        context = super(AuthorDetail, self).get_context_data(**kwargs)
        form_class = self.get_form_class()
        context['form'] = self.get_form(form_class)
        return context

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()
        self.object = self.get_object()
        form_class = self.get_form_class()
        form = self.get_form(form_class)
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        # Here, we would record the user's interest using the message
        # passed in form.cleaned_data['message']
        return super(AuthorDetail, self).form_valid(form)
```

get_success_url() is just providing somewhere to redirect to, which gets used in the default implementation of form_valid(). We have to provide our own post() as noted earlier, and override get_context_data() to make the Form available in the context data.

A better solution

It should be obvious that the number of subtle interactions between FormMixin and DetailView is already testing our ability to manage things. It's unlikely you'd want to write this kind of class yourself.

In this case, it would be fairly easy to just write the post() method yourself, keeping DetailView as the only generic functionality, although writing Form handling code involves a lot of duplication.

Alternatively, it would still be easier than the above approach to have a separate view for processing the form, which could use FormView distinct from DetailView without concerns.

An alternative better solution

What we're really trying to do here is to use two different class based views from the same URL. So why not do just that? We have a very clear division here: GET requests should get the DetailView (with the Form added to the context data), and POST requests should get the FormView. Let's set up those views first.

The AuthorDisplay view is almost the same as when we first introduced AuthorDetail; we have to write our own get_context_data() to make the AuthorInterestForm available to the template. We'll skip the get_object() override from before for clarity.

```
from django.views.generic import DetailView
from django import forms
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDisplay(DetailView):
    model = Author

    def get_context_data(self, **kwargs):
        context = super(AuthorDisplay, self).get_context_data(**kwargs)
        context['form'] = AuthorInterestForm()
        return context
```

Then the AuthorInterest is a simple FormView, but we have to bring in SingleObjectMixin so we can find the author we're talking about, and we have to remember to set template_name to ensure that form errors will render the same template as AuthorDisplay is using on GET.

```
from django.core.urlresolvers import reverse
from django.http import HttpResponseForbidden
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin

class AuthorInterest(SingleObjectMixin, FormView):
    template_name = 'books/author_detail.html'
    form_class = AuthorInterestForm
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()
        self.object = self.get_object()
        return super(AuthorInterest, self).post(request, *args, **kwargs)

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.object.pk})
```

Finally we bring this together in a new AuthorDetail view. We already know that calling as_view() on a

class-based view gives us something that behaves exactly like a function based view, so we can do that at the point we choose between the two subviews.
You can of course pass through keyword arguments to as_view() in the same way you would in your URLconf, such as if you wanted theAuthorInterest behavior to also appear at another URL but using a different template.

```
from django.views.generic import View

class AuthorDetail(View):

    def get(self, request, *args, **kwargs):
        view = AuthorDisplay.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterest.as_view()
        return view(request, *args, **kwargs)
```

This approach can also be used with any other generic class-based views or your own class-based views inheriting directly from View orTemplateView, as it keeps the different views as separate as possible.

More than just HTML

Where class based views shine is when you want to do the same thing many times. Suppose you're writing an API, and every view should return JSON instead of rendered HTML.
We can create a mixin class to use in all of our views, handling the conversion to JSON once.
For example, a simple JSON mixin might look something like this:

```
import json
from django.http import HttpResponse

class JSONResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return HttpResponse(
            self.convert_context_to_json(context),
            content_type='application/json',
            **response_kwargs
        )

    def convert_context_to_json(self, context):
        "Convert the context dictionary into a JSON object"
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return json.dumps(context)
```

Note
Check out the Serializing Django objects documentation for more information on how to correctly transform Django models and querysets into JSON.
This mixin provides a render_to_json_response() method with the same signature as render_to_response().

To use it, we simply need to mix it into a TemplateView for example, and override render_to_response() to call render_to_json_response() instead:

```
from django.views.generic import TemplateView

class JSONView(JSONResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

Equally we could use our mixin with one of the generic views. We can make our own version of DetailView by mixing JSONResponseMixin with the django.views.generic.detail.BaseDetailView – (the DetailView before template rendering behavior has been mixed in):

```
from django.views.generic.detail import BaseDetailView

class JSONDetailView(JSONResponseMixin, BaseDetailView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

This view can then be deployed in the same way as any other DetailView, with exactly the same behavior – except for the format of the response.

If you want to be really adventurous, you could even mix a DetailView subclass that is able to return both HTML and JSON content, depending on some property of the HTTP request, such as a query argument or a HTTP header. Just mix in both the JSONResponseMixin and aSingleObjectTemplateResponseMixin, and override the implementation of render_to_response() to defer to the appropriate rendering method depending on the type of response that the user requested:

```
from django.views.generic.detail import SingleObjectTemplateResponseMixin

class HybridDetailView(JSONResponseMixin, SingleObjectTemplateResponseMixin, BaseDetailView):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get('format') == 'json':
            return self.render_to_json_response(context)
        else:
            return super(HybridDetailView, self).render_to_response(context)
```

Because of the way that Python resolves method overloading, the call to super(HybridDetailView, self).render_to_response(context)ends up calling the render_to_response() implementation of TemplateResponseMixin.