

# Replacing Django's View function by re-usable Objects

**Author :** Frederic De Zorzi  
**Contact :** [fredz@nospam.pimentech.fr](mailto:fredz@nospam.pimentech.fr)  
**Revision :** 2  
**Date :** 2010-07-02  
**Copyright :** This document has been placed in the public domain.  
**Tags :** django

## Abstract

Never been frustrated doing functional programming in your [django views](#) ? This little module is intended to help you use objects for your views :

- Replace *urls parameters* and *template context* by class attributes.
- Use inheritance to group common task, do specializations, common decorators...
- Define template tags in your objects, call instance methods.
- Define your re-usable library.

This code is now quite complete and we use it in production environment.

## Hello World comparison

Consider this example (classic way) :

### urls.py

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

urlpatterns = patterns(
    '',
    (r'^hello/(?P<times>\d+)/$', hello),
)
```

### views.py

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

def hello(request, times=10):
    name = request.user and request.user.username or 'Pony'
    times = range(int(times))
    return render_to_response('hello.html',
                              { 'times' : times, 'name' : name },
                              context_instance=RequestContext(request))
```

### hello.html

```
{% for i in times %}
Hello {{ name }}
{% endfor %}
```

**Now with ViewsObject :****urls.py**

```

from django.conf.urls.defaults import *
from django_pimentech.viewsojects import ObjectCaller
from views import Hello

urlpatterns = patterns(
    '',
    (r'^hello/(?P<times>\d+)/$', ObjectCaller(Hello)),
)

```

**views.py**

```

from django_pimentech.viewsojects import BaseView

class Hello(BaseView):
    template = 'hello.html'
    name = None
    times = 10

    def fill_context(self):
        self.times = range(int(self.times))
        self.name = self.request.user and self.request.user.username or 'Pony'

```

Besides the object syntax, the main idea of this system is that url parameters and context are **regrouped in attributes and method classes** : each public attribute or method will be accessible by the template.

That's it. We have implemented all the views function capabilities, as you will see below.

## Usage

### URLs

```

from django.conf.urls.defaults import *
from django_pimentech.viewsojects import ObjectCaller
from views import MyView

urlpatterns = patterns(
    '',
    (r'^test/(?P<param1>\w+)/(?P<param2>\w+)/(?P<paramn>\w+)/$', ObjectCaller(MyView))
)

```

- **Each parameter will be a MyView instance attribute.**
- The *ObjectCaller* class is important here : when you import the urls module, the ObjectCaller is initialized (`__init__` method). Then, each time the url above is called, the ObjectCaller instance is called, like a function (`__call__` method). If you put “MyView” directly on urls module, the same instance will be shared by multiple http clients, be warned !

### View definition and context

```

from django_pimentech.viewsojects import BaseView

class MyView(BaseView):
    template = 'my_template.html'

    param1 = default_value
    param2 = None
    ...
    paramn = None

```

```
def fill_context(self):
    self.param2 = self.something()
```

- Define the *template* attribute if you want to render a template. You can also overload the *get\_template* method (see [source](#)).
- Each attribute defined in your class (here param1..n) defined from the first BaseView inheritance will be integrated in the request context<sup>1</sup>.
- you can access to the request variable with self.request

## Inheritance

Sure ! In most cases, it is better to overload *fill\_context* method, and reserve *\_\_call\_\_* method overload for decorators and authentication related operations.

## Decorators

You have to decorate the *\_\_call\_\_* BaseView method, decorated with the special “on\_method” decorator

```
@on_method(login_required)
def __call__(self, request, *args, **kwargs):
    self.authenticated_user = request.user
    self.employe = self.authenticated_user.get_profile()
    if kwargs.has_key('note'):
        kwargs['note'] = self.get_note_with_perms(kwargs['note'])

    return super(ActiveMemberRequiredView, self).__call__(request, *args, **kwargs)
```

## Template tags and filters as methods

The instance itself is also passed in request context, as *self* attribute. With *self* attribute, you can call any public method that doesn't require parameters

```
{{ self.my_method }}
```

If your method has parameters, you can use the *call* template filter automatically defined when you import viewsobjects

```
{% call "toto" "popo" forloop.counter %}
```

will call the *toto* method of your viewobject instance with the parameters “popo” and forloop.counter. You can also easily define template tags like this

```
def module_note(self, note):
    return self.inclusion_tag("note.html", { 'new': False,
                                           'note': note})
```

called in your template with

```
{% call "module_note" myvar %}
```

Note that the template tag is only compiled once at it's first use.

## HttpResponse

In your *fill\_context* method, you can return a HttpResponse (a 404 for example), the “render\_to\_response” mechanism will be automatically shorted.

<sup>1</sup>The request context parameters list is defined in module import stage (see meta-class in [source](#)), for better performance.

## JSON response

A little shortcut is provided if you have to render *JSON* instead of a template : simply overload the *render* method like this

```
def render(self):
    return self.render_json()
```

With this configuration, all your context will be returned in a JSON *HttpResponse* string. If you want to return only some variables, pass your structure to the *render\_json* method

```
def render(self):
    return self.render_json(self.my_result)
```

## Installation

– get PimenTech libcommonDjango :

```
svn checkout http://svn.pimentech.org/pimentech/libcommonDjango
```

– install it with “make install”<sup>2</sup>

Trac access : <http://trac.pimentech.org/pimentech/browser/libcommonDjango>

## Source

You can also directly get the source [here](#).

```
# -*- coding: utf-8 -*-
from django.http import HttpResponse
from django.shortcuts import render_to_response
from django.template import defaulttags, RequestContext, TemplateSyntaxError, Node
from django.template.loader import get_template, select_template

try:
    from django.utils import simplejson
    from django.core.serializers.json import DjangoJSONEncoder
except ImportError:
    pass # does'nt work in 0.97
import types

class ObjectCaller:
    def __init__(self, klass, *args, **kwargs):
        self.klass = klass
        self.args = args
        self.kwargs = kwargs
    def __call__(self, *args, **kwargs):
        obj = self.klass(*self.args, **self.kwargs)
        return obj(*args, **kwargs)

class MetaViewClass(type):
    def __new__(meta, classname, bases, classDict):
        if classname != 'BaseView':
            classDict['_context_attrs'] = {}
            for base in bases:
```

---

<sup>2</sup>or *python setup.py install* in libcommonDjango dir

```

        if base.__dict__.has_key('_context_attrs'):
            classDict['_context_attrs'].update(base.__context_attrs)
    for attrname, value in classDict.items():
        if not attrname.startswith('_') and not type(value) in (types.Class, types.Module):
            classDict['_context_attrs'][attrname] = attrname

    return type.__new__(meta, classname, bases, classDict)

class BaseView(object):
    __metaclass__ = MetaViewClass

    template = ''
    request = None
    context = None

    def __init__(self, *args, **kwargs):
        self.context = {}
        self.template = kwargs.get('template') or self.template
        self.context.update(kwargs.get('extra_context', {}))

    def get_template(self):
        return self.template

    def render_html(self):
        return render_to_response(self.get_template(),
                                   self.context,
                                   context_instance=RequestContext(self.request))

    def render_json(self, to_pack=None):
        response = HttpResponse(mimetype='application/x-javascript')
        if to_pack is None:
            del self.context['self']
        simplejson.dump(to_pack or self.context, response, ensure_ascii=False, cls=JSONEncoder)
        return response

    def render(self):
        return self.render_html()

    def fill_extra_context(self):
        # DEPRECATED
        pass

    def fill_context(self):
        """Defined in subclasses"""
        pass

    def _fill_context_with_attrs(self):
        for attrname in self._context_attrs.keys():
            if not self.context.has_key(attrname):
                self.context[attrname] = getattr(self, attrname)

    def process(self):
        response = self.fill_context()
        if hasattr(response, 'status_code'):
            # http response

```

```

        return response
    response = self.fill_extra_context()
    if hasattr(response, 'status_code'):
        # http response
        return response
    self._fill_context_with_attrs()
    return self.render()

def set_initial_attrs(self, request, kwargs):
    self.request = request
    self.context['self'] = self
    for var, value in kwargs.items():
        setattr(self, var, value)

def __call__(self, request, *args, **kwargs):
    self.set_initial_attrs(request, kwargs)
    return self.process()

__compiled_template__ = None
def inclusion_tag(self, file_name, extra_context=None):
    if self.__compiled_template__ is None:
        self.__compiled_template__ = {}
    nodelist = self.__compiled_template__.get(file_name)
    if nodelist is None:
        if not isinstance(file_name, basestring) and is_iterable(file_name):
            t = select_template(file_name)
        else:
            t = get_template(file_name)
        nodelist = self.__compiled_template__[file_name] = t.nodelist

    if extra_context:
        context = self.context.copy()
        context.update(extra_context)
    else:
        context = self.context
    return nodelist.render(Context(context))

# Thanks to Todd Reed from http://www.toddreed.name/content/django-view-class/
def on_method(function_decorator):
    def decorate_method(unbound_method):

        def method_proxy(self, *args, **kwargs):
            def f(*a, **kw):
                return unbound_method(self, *a, **kw)

            return function_decorator(f)(*args, **kwargs)

        return method_proxy

    return decorate_method

```

```

# inspired by http://code.djangoproject.com/wiki/CallTag
class CallNode(Node):
    def __init__(self, method_name, *args, **kwargs):
        self.method_name = method_name
        self.args = args
        self.kwargs = kwargs

    def render(self, context):
        method_name = self.method_name.resolve(context)
        method = getattr(context['self'], method_name)
        d = {}
        args = d['args'] = []
        kwargs = d['kwargs'] = {}
        args = [ arg.resolve(context) for arg in self.args ]
        return method(*args)

def do_call(parser, token):
    bits = token.contents.split()
    if len(bits) < 2:
        raise TemplateSyntaxError, "%r tag takes at least one argument" % bits[0]

    method_name = parser.compile_filter(bits[1])
    if method_name.token[1] == '_':
        raise TemplateSyntaxError, "Private method calls are not allowed: %r" % method_name

    args = [ parser.compile_filter(arg) for arg in bits[2:] ]

    return CallNode(method_name, *args)

defaulttags.register.tag('call', do_call)
# Perhaps the line above is not very academic

```

## Note

We would be pleased to receive your feedbacks, corrections and improvement suggestions.