
aino-utkik Documentation

Release 0.7.1

Mikko Hellsing

October 01, 2015

1	Quickstart	3
1.1	Requirements, Installation & Configuration	3
1.2	Usage	3
2	Introduction	7
2.1	The past	7
2.2	The future	8
2.3	I can't stop	9
2.4	Who are you?	10
3	Comparing implementations	11
4	Reference	15
4.1	View	15

aino-utkik provides minimalistic class based views for Django focusing on common usage, readability and convenience.

Quickstart

1.1 Requirements, Installation & Configuration

You will need at least Python 2.5+ and Django 1.0+

Firstly install the package from pypi:

```
pip install aino-utkik
```

Secondly, you need to setup the dispatcher by replacing all imports in all your urls.py. Before:

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

After:

```
from utkik.dispatch import
```

Anything after the import on the same row should be kept intact. If you are brave you can place your self in your project root and issue:

```
find -name urls.py | xargs perl -p -i -e 's/from\ django\.conf\.urls\.defaults/from utkik.dispatch/g'
```

1.2 Usage

1.2.1 Dispatcher

Now you are ready to use the utkik dispatcher. This allows you to reference a class based view based from Django 1.3 Class-based generic views or `utkik.View` as a string in your urls.py (or any other class based view that has an entry point method called `dispatch`). Assuming that your class resides in `myapp.views.Home` and that `myapp` is in `INSTALLED_APPS` you may reference it as follows:

```
from utkik.dispatch import *

urlpatterns = patterns('',
    (r'^$', 'myapp.Home'),
)
```

Alternatively you can reference it with a more accurate string. You can always reference a class based view even if it's not in `INSTALLED_APPS` this way:

```
from utkik.dispatch import *

urlpatterns = patterns('',
    (r'^$', 'myapp.views.Home'),
)
```

1.2.2 utkik.View

I will describe the main ideas here and if you have the time please look through the [source code](#), it is really small.

Firstly the *handler* for an allowed request is defined as a method that has the same name as the HTTP request method, only lower cased. So a GET request will try to call a get method on the class and a POST method a post and so on. A method is allowed only if the class has a corresponding lower case named attribute on the class. But because not everyone keeps those method names in their head there is an additional attribute in the class that controls if the method should be allowed or not and that is `methods`. By default this is set to `['GET', 'POST', 'PUT', 'DELETE']` thus allowing only GET, POST, PUT and DELETE. If you want to allow anything else you first have to add that method name to this list and then create a method with the lower case name on the class.

So basically you do all your stuff in the handler or conduct it from there, for example `MyView.get`. The handler gets passed any additional arguments that is parsed from the `urls.py` but not the request. The request object is accessed through the class instance it self `self.request`. So your get handler might start something like:

```
def get(self, slug):
    ...
```

At the end of the handler you can either return a valid `HttpResponse` or you don't return anything. If you don't return anything the `self.render` method will be called. This method renders the first existing template returned from `get_template_names()` with `self.get_context_data()` as context data and returns the result. Unless `template_name` nor `ajax_template_name` is defined `get_template_names()` returns a list of automatically computed template names. This is how this is done: First we try to figure out the current `app_label` then we use the current name of the view class but using lower-case letters and underscores thus `ProductList` will become `product_list` and so on. From these variables we put together a path for the template: `<< app_label >>/<< un-cameled class name >>.html` and for ajax calls this is `<< app_label >>/<< un-cameled class name >>.ajax.html`. You have the option to override this, although I discourage doing so. The class properties are: `template_name` and `ajax_template_name`.

But wait there is more! In your view you can reference an object representing the context as `self.c`. You can set stuff to the context as follows:

```
self.c.news = get_object_or_404(News.objects, slug=slug)
```

The `self.get_context_data` by default returns this context object as a dictionary. Adding a decorator is a no brainer too, just add it to the `self.decorators` list. If you want to add a decorator for GET but not for POST, that is *a specific decorator per handler* you can use yet another decorator `utkik.decorators.handler_decorator`. This decorator accepts normal view function decorators like `django.contrib.auth.decorators.login_required`. Example:

```
from django.contrib.auth.decorators import login_required
from functional import wraps
from utkik.decorators import handler_decorator, require_ajax
from utkik import View, JsonResponse

def mydecorator(f):
    """function view decorator"""
    @wraps(f):
    def wrapper(request, *args, **kwargs):
        if not request.user.email.endswith('@aino.se'):
            ...
```



```
        return HttpResponse(status=402)
    return f(request, *args, **kwargs)
return wrapper

class Home(View):
    @handler_decorator(login_required, mydecorator)
    def get(self):
        pass

    @handler_decorator(require_ajax):
    def post(self):
        return JsonResponse({ "message": "rock my pony" })
```

Now, lets bake another simple view example:

```
from django.contrib.auth.decorators import login_required
from utkik import View
from news.models import News

class NewsDetail(View):
    decorators = [ login_required ]

    def get(self, slug):
        self.c.news = get_object_or_404(News.objects, slug=slug)
```

That is all there is to it! You are not returning anything from the handler and thus letting `self.render` do the work.

For more please [read the code](#) and see the [examples](#).

Introduction

2.1 The past

We are lazy and we should be. Class based views has been a long time itch for us Django users. There are a lot good use-cases for using classes for your views instead of functions. The first that come in to mind is probably inheritance, define a view and then re-use some of that code for another view without having to completely rewrite it, that's great right? For example:

```
class MyBaseView(object):
    def get_queryset(self):
        raise NotImplemented

    def do_something_useful(self):
        qs = self.get_queryset()
        ...

class MyView(MyBaseView):
    def get_queryset(self):
        return MyModel.objects.all()
```

For Django users this was a bit hard to solve since the view had to be a callable in the current infrastructure. Some people started out creating class based views using the `__call__` method of a class like this:

```
class OopsView(object):
    def __call__(self, request):
        self.request = request

    def render(self):
        return response
```

And with something like this in `urls.py`:

```
from django.conf.urls.defaults import *
from oops.views import OopsView

urlpatterns = patterns('',
    (r'^$', OopsView()),
)
```

The problem with this approach is that you are creating an instance of the class in `urls` and then you are setting a state to it in `self.request = request` effectively making the view one instance sharing it's state across requests. All kinds of solutions exists to solve this problem, they can be divided into two different approaches, the first that does not store state that changes across requests (used by `django.contrib.admin`) which requires you to pass things

like request around and the second implementation is to create a new instance on every request. The latter being more safe and you do not have to worry about sharing state since every request will have their own class instance. Django adopted class based views for 1.3 generic views using the second approach. The way its done is by having a class method on the view class that returns a function (callable) that returns a class instance and calling the instances entry point (dispatch). To use a class based view subclassed from `django.views.generic.View` you now call the class method `as_view` in your `urls.py`:

```
from django.conf.urls.defaults import *
from myapp.views import MyView

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

This works fine except for a few things:

1. If there is an import error in `urls.py` you will have a very hard time to debug it.
2. The `urls.py` can get really cluttered with view imports.
3. Writing `.as_view()` that many times is really boring and is a weird redundant suffix.
4. We add unnecessary boilerplate code to the views.

2.2 The future

What I would like to be able to do is more something like:

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

urlpatterns = patterns('',
    (r'^$', 'myapp.MyView'),
)
```

Now `myapp.MyView` is not a correct “python string notation” as would probably be `myapp.views.MyView`, did I say I was lazy? After all a string is not python is it? This would solve all of the things listed above:

1. We would have lazy imports, working out the view first when it is called. Getting a proper traceback in case something goes wrong, well hopefully.
2. We don’t need to do all those imports into `urls.py`
3. Less writing is good as long as it is explicit.
4. Writing a view is as simple as subclassing object and adding a dispatch method.

So how can we achieve something like this?

1. If we are good with using `"myapp.views.MyView"` then we can hack something together using metaclasses and what not in the view class. Or we can change the handler.
2. We change the behaviour of the `django.conf.urls.defaults.url` function.
3. Change the handler to not just accept a callable in addition to #2.

As for changing the handler, let’s just say that I rather not. I figured #3 would be the best solution but that is more work than #2, did I say I was lazy? `utkik.dispatch.url` is class based view aware. The modification was not very big nor hard, have a look in the source code. To outline the basic idea, we create a class that wraps the view, be it a class view or a function view string notated or not and when that class instance is called (`__call__`) on from the django handler it will either return the view function call or create a new instance of the class view and calling its entry point (dispatch).

2.3 I can't stop

It's great now we have a nice way to route to class based views. The next topic is class based views. What do we want them to do and how do we want to work with them? Here is an unordered in terms of importance wish list of things I would like to see:

1. Catch bad requests and return a proper error code.
2. Render a certain template with context using very little code.
3. Make it easy to update the current context for the template rendering.
4. Have sensible hooks for subclassing.
5. Subclasses should be easy to read and follow.
6. They should be very convenient but allow for special cases without breaking a sweat.
7. Reading the source code should be easy.
8. Applying decorators should be easy and inheritable.

Of course I am just listing all those things that match what my current implementation has, I just do that to make me look good. Django 1.3 class based generic views certainly does some of these things but unfortunately some out right not. This is just speaking from a general class based view point not taking the generic part into account. Beware that this means I am not totally fair since the generic views will have problems to share my goals with the other goals set for them. So how does Django 1.3 class-based generic views stack up?

1. Catch bad requests and return a proper error code.

Yes, but only partially, the default behaviour is to allow every HTTP method if it has such a lowercase attribute of the class. I would say it is a rare use-case for anything else but GET/POST/PUT/DELETE, something could definitely go wrong there. It does not check for ajax at all.

2. Render a certain template with context using very little code.

This it can achieve very well, although you need the docs around to remember the method and variable names to set.

3. Make it easy to update the current context for the template rendering.

Well it's not hard but still not quite there, like in `utkik.View` there is also a method called `get_context_data` This is probably thought of where you should get all your context data. In my mind that is mostly what a view does in a typical case, It collects data for the context that it used to render the template with. I guess this is where most of your views code will end up unless you do something creative. `utkik.View` used this method to do the final getting of data but it is not intended to be where it is all collected and calculated.

4. Have sensible hooks for subclassing.

I think the dispatch method could be divided into smaller parts to better allow subclassing, for the more specialized generic cases I do not know.

5. Subclasses should be easy to read and follow.

See 6

6. They should be very convenient but allow for special cases without breaking a sweat.

Granted, they are convenient, but they still suffer from what we had from the old function based generic views. When you want to do something a little different, it's hard, so hard you need to read the source. When you read the source you also notice its full of nice mixins. Mixins are great but it makes it very hard to follow and you sort of have to construct that final class in your head, tricky for sure. Anyway, once you managed to get that class doing what you wanted you realize that it is very hard to follow as well, let alone remember.

7. Reading the source code should be easy.

It is just not because of all the generalizations and mixins.

8. Applying decorators should be easy and inheritable.

You can do this in two ways basically, urls or views:

```
# urls.py (not inheritable)
login_required(MyView.as_view())

# views.py
@method_decorator(login_required)
def dispatch(self, *args, **kwargs):
    return super(MyView, self).dispatch(*args, **kwargs)
```

None of which are I think is very nice.

2.4 Who are you?

I am a minimalist class based view base class for Django (MCBVBCFD) as seen on [github](#)

Comparing implementations

The following are simple, quite common views from a real world application.

Django 1.3 Class-based generic views:

```
class ArtistDetail(DetailView):
    context_object_name = 'artist'
    queryset = Artist.publ.all()
```

utkik.View:

```
class ArtistDetail(View):
    def get(self, slug):
        self.c.artist = get_object_or_404(Artist.publ, slug=slug)
```

Note: The following two list views are written a little differently regarding the context collection to reflect more real world usage.

Django 1.3 Class-based generic views:

```
class ArtistList(TemplateView):
    template_name = 'artists/artist_list.html'

    def get_context_data(self, slug=None, tag=None):
        genre = get_object_or_404(Genre.objects, slug=slug)
        artists = Artist.publ.filter(genres=genre).order_by('?')
        tags = Tag.objects\
            .filter(artist__genres=genre, artist__active=True)\
            .annotate(num_tags=Count('artist'))\
            .filter(num_tags__gte=2)\
            .order_by('-num_tags')\
            .distinct()

        return {
            'artists': artists,
            'tags': tags,
            'selected_genre': genre,
        }
```

utkik.View:

```
class ArtistList(View):
    def get(self, slug=None, tag=None):
        self.c.selected_genre = get_object_or_404(Genre.objects, slug=slug)
        self.c.artists = Artist.publ.filter(genres=self.c.selected_genre).order_by('?')
```

```
self.c.tags = Tag.objects\
    .filter(artist__genres=self.c.selected_genre, artist__active=True)\
    .annotate(num_tags=Count('artist'))\
    .filter(num_tags__gte=2)\
    .order_by('-num_tags')\
    .distinct()
```

Django 1.3 Class-based generic views:

```
class MoreArtists (TemplateView):
    template_name = 'artists/inc/artists_in_focus.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseForbidden()
        return super(MoreArtists, self).dispatch(request, *args, **kwargs)

    def get_context_data(self):
        return { 'artists': Artist.publ.all().order_by('?')[:6] }
```

utkik.View:

```
class ArtistsInFocus (View):
    decorators = [ requires_ajax ]

    def get(self):
        self.c.artists = Artist.publ.all().order_by('?')[:6]
```

Django 1.3 Class-based generic views:

```
class ArtistLogin (FormView):
    form_class = ArtistLoginForm
    template_name = 'artists/artist_login.html'

    def get(self, request, *args, **kwargs):
        request.session.set_test_cookie()
        return super(ArtistLogin, self).get(request, *args, **kwargs)

    def form_valid(self, form):
        login(self.request, form.artist)
        return HttpResponseRedirect(reverse('artist_mypage'))

    def get_form_kwargs(self):
        kwargs = super(ArtistLogin, self).get_form_kwargs()
        kwargs['request'] = self.request
        return kwargs
```

utkik.View:

```
class ArtistLogin (View):
    def setup(self):
        self.c.form = ArtistLoginForm(
            request=self.request, data=self.request.POST or None)

    def get(self):
        self.request.session.set_test_cookie()

    def post(self):
        if self.c.form.is_valid():
```



```
login(self.request, self.c.form.artist)
return HttpResponseRedirect(reverse('artist_mypage'))
```

Django 1.3 Class-based generic views:

```
from django.conf.urls.defaults import *
from artists.views import ArtistDetail, ArtistList, ArtistTagList, MoreArtists, ArtistSearch, ArtistMyPage

urlpatterns = patterns('',
    url(r'^min-sida/$', ArtistMyPage.as_view(), name='artist_mypage'),
    url(r'^artister/$', MoreArtists.as_view(), name='artist_more'),
    url(r'^artister/sok/$', ArtistSearch.as_view(), name='artist_search'),
    url(r'^artister/(?P<slug>[-\w]+)/$', ArtistList.as_view(), name='artist_genre_list'),
    url(r'^artister/(?P<slug>[-\w]+)/(?P<tag>.*)/$', ArtistTagList.as_view(), name='artist_genre_tag'),
    url(r'^(?P<slug>[-\w]+)/$', ArtistDetail.as_view(), name='artist_detail'),
)
```

utkik.View:

```
from utkik.dispatch import *

urlpatterns = patterns('',
    url(r'^min-sida/$', 'artists.ArtistMyPage', name='artist_mypage'),
    url(r'^artister/$', 'artists.MoreArtists', name='artist_more'),
    url(r'^artister/sok/$', 'artists.ArtistSearch', name='artist_search'),
    url(r'^artister/(?P<slug>[-\w]+)/$', 'artists.ArtistList', name='artist_genre_list'),
    url(r'^artister/(?P<slug>[-\w]+)/(?P<tag>.*)/$', 'artists.ArtistTagList', name='artist_genre_tag'),
    url(r'^(?P<slug>[-\w]+)/$', 'artists.ArtistDetail', name='artist_detail'),
)
```

Note: You can of course use the utkik dispatcher for Django 1.3 Class-based generic views too.

Reference

4.1 View

4.1.1 attributes

methods

This should contain a list of allowed HTTP methods for the view. The names should be upper-case just as they are named in the HTTP specification. If the class has a method of the same name but in lower case this will be called to when a request is made.

decorators

A list of decorators applied to `get_response`.

template_name

This is the template that the `render()` will firstly try to render to. If you don't set this or this template is not found `get_template_names()` will also return an automatically computed template name for you as: `<< app_label >>/<< un-cameled class name >>.html`.

ajax_template_name

This does exactly as `template_name` but for ajax calls and the computed template name is: `<< app_label >>/<< un-cameled class name >>.ajax.html`.

4.1.2 methods

TODO make docstrings go here