django-baseviews Documentation

Release 0.5

Brandon Konkle

March 08, 2014

Contents:

CHAPTER 1

Installation

Use pip to install the module:

```
$ pip install django-baseviews
```

Then simply import it for use in your views:

```
from baseviews.views import BasicView
```

CHAPTER 2

Writing Views

2.1 Basic Views

The simplest views can be handled by creating a subclass of `BasicView`, defining the `template` attribute, and implementing the `get_context` method.

```
from baseviews.views import BasicView
from lol.models import Cheezburger

class LolHome(BasicView):
    template = 'lol/home.html'

    def get_context(self):
        return {'burgers': Cheezburger.objects.i_can_has()}
```

2.2 Custom MIME type

The MIME type defaults to the value of the `DEFAULT_CONTENT_TYPE` setting. This can be overriden by defining the content_type attribute:

```
from baseviews.views import BasicView
from lol.models import Cheezburger

class GoogleSiteMap(BasicView):
    template = 'sitemap.xml'
    content_type = 'application/xml'
```

2.3 Caching the Context

If you'd like to cache the context through the low-level cache API, add the `cache_key` and `cache_time` attributes and override the `cached_context` method instead of the `get_context` method. Additionally, you can override `uncached_context` to add context that shouldn't be cached. If `cache_time` isn't set, it defaults to the arbitrary length of 5 minutes.

```
class LolHome(BasicView):
    template = 'lol/home.html'
    cache_key = 'lol_home'
    cache_time = 60*20 # 20 minutes
```

```
    def cached_context(self):
        return {'burgers': Cheezburger.objects.i_can_has()}
```

The `cache_key` attribute can include string formatting, which you can populate by overriding the `get_cache_key` method:

```
class LolDetail(BasicView):
    template = 'lol/detail.html'
    cache_key = 'lol_detail:%s'
    cache_time = 60*20 # 20 minutes

    def __init__(self, request, lol_slug):
        self.lol = Lol.objects.get(slug=lol_slug)
        super(LolDetail, self).__init__(request)

    def get_cache_key(self):
        return self.cache_key % self.lol.slug
```

2.4 Ajax Views

The `AjaxView` class is a subclass of `BasicView` that takes the context and uses simplejson to dump it to a JSON object. If the view is not requested via Ajax, it raises an Http404 exception.

2.5 Decorators

Built-in decorators such as login_required don't work by default with class-based views. This is because the first argument passed to the decorator is the class instance, not the request object.

To decorate a class-based view, simply use the helper `django.utils.decorators.method_decorator` on the `__new__` method like this:

```
from django.utils.decorators import method_decorator
from django.contrib.auth.decorators import login_required
from baseviews.views import BasicView

class BucketFinder(BasicView):
    template = 'lol/wheres_mah_bucket.html'

    @method_decorator(login_required)
    def __new__(cls, *args, **kwargs):
        return super(BucketFinder, cls).__new__(cls, *args, **kwargs)
```

2.6 Form Views

Form processing can be simplified with a subclass of the `FormView` class. Define an extra attribute called `form_class` and set it to the form you'd like to use, and define an attribute called `success_url` with the name of the url to be redirected to after successful form processing. You can also override the `get_success_url` method to provide a dynamic success url.

The most basic processing can be handled without any further effort. `FormView` will get the form and add it to the context, and if the request method is POST it will attempt to validate and save it.

If you would like to do more, you can extend the `get_form` and `process_form` methods:

```
class KittehView(FormView):
    template = 'lol/kitteh.html'
    form_class = KittehForm

    def __init__(self, request, kitteh_slug):
        self.kitteh = get_object_or_404(Kitteh, slug=kitteh_slug)
        super(KittehView, self).__init__(request)

    def get_form(self):
        self.form_options = {'request': self.request,
                             'kitteh': self.kitteh}
        return super(KittehView, self).get_form()

    def process_form(self):
        if self.request.POST.get('edit', False):
            if self.form.is_valid():
                self.form.save()
                return redirect(self.get_success_url())
        elif self.request.POST.get('delete', False):
            self.kitteh.delete()
            return redirect('kitteh_deleted')

    def get_success_url(self):
        return reverse('kitteh_edited', args=[self.kitteh.slug])
```

2.7 Views with Multiple Forms

If you need multiple forms in one view, use MultiFormView. This is a subclass of FormView that allows you to provide `form_classes` dict as an attribute on the class, mapping form names to form classes. The form names will be used as the keys to form instances, and each form name will be turned into a context variable providing the form instances to your template.

```
class MonorailCatTicketsView(MultiFormView):
    template = 'lol/monorail_tickets.html'
    form_classes = {'kitteh_form': KittehForm,
                    'payment_form': PaymentForm}

    def __init__(self, request, kitteh_slug):
        self.kitteh = get_object_or_404(Kitteh, slug=kitteh_slug)
        super(MonorailCatTicketsView, self).__init__(request)

    def get_form(self):
        self.form_options['kitteh_form'] = {'request': self.request,
                                            'kitteh': self.kitteh}
        self.form_options['payment_form'] = {'user': self.request.user}
        return super(MonorailCatTicketsView, self).get_form()

    def get_success_url(self):
        return reverse('monorail_cat_thanks_you', args=[self.kitteh.slug])
```

2.8 Mapping the Views to URLs

In order to make the use of class attributes safe, baseviews overrides the __new__ method on the class. This means that you can simply map the url pattern directly to the class:

```
from lol import views

urlpatterns = patterns('',
    url(r'^$', views.LolHome, name='lol_home'),
)
```

CHAPTER 3

View Reference

This document describes the details of the view classes that baseviews provides.

3.1 BasicView

class BasicView
A basic view that renders context to template, optionally caching the context.

cache_key
Set this to a string to enable caching. An easy way to use a dynamic cache key is to include string formatting specifiers in the string, which you can then convert in the `get_cache_key` method.

cache_time
Controls the time, in seconds, to use for in caching. It defaults to the arbitrary value of 5 minutes.

content_type
Provides an opportunity to customize the mimetype used in the `render` method. Defaults to `settings.DEFAULT_CONTENT_TYPE`.

get_context()
This returns the context that is passed to the `render` method. Override this method to provide context to your template.

cached_context()
If `get_context` is not overridden, it will call this method to retrieve the context. If the `cache_key` attribute on the view class is set, then it will cache this context.

uncached_context()
After it retrieves `cached_context`, the `get_context` method calls this and updates the context dict with the context this method returns. The context will not be cached.

get_cache_key()
By default, this simply returns the `cache_key` attribute from the view class. The point of this is to give you a chance to dynamically generate the cache key based on the request, including things like object id's or slugs in the key that is returned by this method.

get_template()
This defaults to the `template` attribute, but the method can be overridden in order to dynamically generate the template based on the request.

render()
Calls `get_template` and `get_context`, and renders the template with the mimetype from the

`content_type` attribute. This can be overridden to customize the rendering, such as outputting to different formats like JSON.

__init__()
Sets the request, args, and kwargs as attributes on the class instance.

__call__()
Returns the results of `render`.

3.2 AjaxView

class AjaxView
A subclass of `BasicView` that returns the context rendered to a JSON object.

content_type
This defaults to *"application/json"*.

__call__()
Checks to make sure that the request is Ajax-based. If not, raises a 404.

render()
Uses simplejson to render the context as a JSON object.

3.3 FormView

class FormView
A subclass of `BasicView` that includes a form in the context and then attempts to process the form if data was provided via POST.

form_class
This is the class of the form that will be instantiated by the view.

success_url
The url that the user will be redirected to after a successful form submission.

uncached_context()
Adds the form instance to the uncached context.

get_form()
If POST data or uploaded files are included in the request, they are added to the `form_options` dict before the `form_class` is instantiated.

process_form()
If the form is valid, this method saves it and then returns a redirect to the `success_url`. Otherwise, it returns `None`, which causes the `__call__` method to call `render` as usual. Data will still be bound to the form after an unsuccessful attempt to process, which allows you to show the error messages in your template.

get_success_url()
By default, it just returs the `success_url` attribute. It can be overridden in your subclass to dynamically determine the url based on the request.

3.4 MultiFormView

class `MultiFormView`

A subclass of `MultiFormView` to handle the processing of more than one form.

`form_classes`

A dict of form names to form classes to be used for the view.

CHAPTER 4

Backwards-Incompatible Changes

4.1 Version 0.5

- **Removed the "from views import *" call from "__init__"** - This was there to provide backwards compatibility for when baseviews was a single file instead of a package. This is not a good practice in general, and it caused problems when trying to implement formal versioning. All instances of `from baseviews import` in your code will need to be replaced with `from baseviews.views import`.

4.2 Version 0.4

- **"view_factory" removed** - With the addition of the `__new__` method override, the class can now used in the url mapping directly. This eliminates the need for a view factory.
- **View args and kwargs handled in "__init__"** - Previously, the view arguments such as `request` and args and kwargs from the url pattern were handled by the `__call__` method. Now, they are (more appropriately) handled by the `__init__` method and the `__call__` method is called without any additional arguments. You'll need to adjust your subclasses accordingly.
- **"decorate" removed** - Jannis Leidel pointed out that Django has an equivalent method decorator built in, at `django.utils.decorators.method_decorator`. This eliminates the need for a custom `decorate` decorator.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*