useful_inkleby Documentation

Release 0.1

Alex Parsons

Apr 30, 2017

Contents

Collection of useful tools for django projects by Alex Parsons. See code on Github.

useful_inkleby.useful_django.models package:

- FlexiModel - Allows manager and queryset methods to be added to a model using @managermethod and @querysetmethod decorators rather than creating custom managers.
- EasyBulkModel - Cleaner bulk creation of objects - store objects to be queued with .queue() and then trigger the save with model.save_queue(). Saves objects in batches. Returns completed objects (nicer than bulk_create).
- StockModelHelpers - Generic methods useful to all models.
- FlexiBulkModel - Combines the above into one class.

useful_inkleby.useful_django.views package:

- IntegratedURLView - Integrates django's url settings directly the view classes rather than keeping them in a separate urls.py.
- BakeView - Handles baking a view into files - expects a function that can feed it arbitrary sets of arguments and a BAKE_LOCATION in the settings.
- FunctionalView - Slimmed down version of class-based views where functional logic is preserved - but class structure used to tidy up common functions.
- MarkDownView - Mixin to read a markdown file into the view.
- ComboView - combines BakeView, IntegratedURLView (most common combo I use).

useful_inkleby.useful_django.fields package:

- JsonBlockField - simple serialisation field that can be handed arbitrary sets of objects for restoration later. Classes can be registered for cleaner serialisation (if you'd like to be able to modify the raw values while stored for instance).

useful_inkleby.files package:

- QuickGrid - compact function to read in spreadsheet and present readable code that translates between spreadsheet headers and internal values (make population scripts nicer).

CHAPTER 1

FlexiModel Usage

```
from useful_inkleby.useful_django.models import FlexiModel, querysetmethod,␣
↪managermethod

class Foo(FlexiModel):

    @querysetmethod
    def bar():
        pass

    @managermethod
    def foobar():
        pass

Foo.objects.foobar()
Foo.objects.all().bar()
```

CHAPTER 2

EasyBulkModel Usage

```
from useful_inkleby.useful_django.models import EasyBulkModel

 class Model(EasyBulkModel):
     pass

 for x in range(10000):
     m = Model(foo=x)
     m.queue()

 Model.save_queue()
```

CHAPTER 3

IntegratedURLView Usage

For the view:

```
class AboutView(IntegratedURLView):
    template = "about.html"
    url_pattern = r'^about'
    url_name = "about_view"

    def view(self,request):
        f = "foo"
        return {'f':f}
```

For project urls.py:

```
from useful_inkleby.useful_django.views import include_view
urlpatterns = [
    url(r'^foo/', include_view('foo.views')), #where foo is the app name
    ]
```

CHAPTER 4

BakeView Usage

settings.py:

```
BAKE_LOCATION = "??" #root to store baked files
```

views.py:

```
class FeatureView(ComboView):
    """
    Feature display view
    """
    template = "feature.html"
    url_pattern = r'^feature/(.*)'
    url_name = "feature_view"
    bake_path = "feature\\{0}.html"

    def bake_args(self):
        """
        arguments to pass into view - a model ref in this case (so equiv to bakery)
        but can also pass non-model arbitrary stuff through.
        """
        features = Feature.objects.all()

        for f in features:
            yield (f.ref,)

    def view(self,request,ref):
        """
        view that is run with the arguments against the template and saved to bake_
↪path
        """
        feature = Feature.objects.get(ref=ref)
        return {"feature":feature}
```

bake.py (script to execute bake):

```
from app.views import FeatureView, bake_static

bake_static()
FeatureView.bake()
```

If combined with IntegratedURLView by using ComboView you can bake a whole app like so:

```
from useful_inkleby.useful_django.views import AppUrl,bake_static
import app.views as views

bake_static()
AppUrl(views).bake()
```

Contents:

useful_inkleby.useful_django.models package

Submodules

useful_inkleby.useful_django.models.flexi module

FlexiModel tidies up adding up methods to models and querysets.

class useful_inkleby.useful_django.models.flexi.**ApplyManagerMethodMeta**
Bases: django.db.models.base.ModelBase

Customise the metaclass to apply a decorator that allows custom manager and queryset methods

class useful_inkleby.useful_django.models.flexi.**CustomRootManager**
Bases: django.db.models.manager.Manager

get_or_none(**args*, ***kwargs*)

use_for_related_fields = True

class useful_inkleby.useful_django.models.flexi.**FlexiModel**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Class for models to inherit to receive correct metaclass that allows the floating decorators

use instead of models.Model

class Meta

abstract = False

useful_inkleby.useful_django.models.flexi.**allow_floating_methods**(*cls*)
Decorator for models that allows functions to be transposed to querysets and managers can decorate models directly - or make a subclass of FlexiModel.

useful_inkleby.useful_django.models.flexi.**managermethod**(*func*)
Decorator for a model method to make it a manager method instead.

will be accessible as model.objects.foo()

“self” will then be the manager object.

self.model - can then be used to access model. self.get_queryset() - to get access to a query

useful_inkleby.useful_django.models.flexi.**querysetmethod**(*func*)
Decorator for a model method to make it apply to the queryset.

Will be accessible as model.objects.all().foo()

"self" will then be the query object.

useful_inkleby.useful_django.models.mixins module

class useful_inkleby.useful_django.models.mixins.**EasyBulkModel**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Bulk Creation Mixin

Mixin to help with creation of large numbers of models with streamlined syntax.

class Meta

abstract = False

classmethod EasyBulkModel.**init_queue**()

EasyBulkModel.**queue**()
Add current object to the class creation queue

classmethod EasyBulkModel.**queue_length**()

classmethod EasyBulkModel.**save_queue**(*safe_creation_rate=1000*, *retrieve=True*)
Saves all objects stored in the class queue in batches.

If retrieve = true (default) will return a list of the saved objects.

classmethod EasyBulkModel.**save_queue_if_count**(*count=1000*)

class useful_inkleby.useful_django.models.mixins.**StockModelHelpers**
Bases: object

common functions useful for all models - diagnostics etc

Module contents

class useful_inkleby.useful_django.models.**FlexiBulkModel**(**args*, ***kwargs*)
Bases: *useful_inkleby.useful_django.models.flexi.FlexiModel*, *useful_inkleby.useful_django.models.mixins.EasyBulkModel*, *useful_inkleby.useful_django.models.mixins.StockModelHelpers*

class Meta

abstract = False

useful_inkleby.useful_django.views package

Submodules

useful_inkleby.useful_django.views.bake module

class useful_inkleby.useful_django.views.bake.**BakeView**
Bases: *useful_inkleby.useful_django.views.functional.LogicalView*

Extends functional view with baking functions.

expects a bake_args() generator that returns a series of different sets of arguments to bake into files.

expects a BAKE_LOCATION - in django settings

render_to_file() - render all possible versions of this view.

classmethod bake(*limit_query=None*, ***kwargs*)
render all versions of this view into a files

bake_args(*limit_query*)
subclass with a generator that feeds all possible arguments into the view

bake_path = ''

render_to_file(*args=None*, *only_absent=False*)
renders this set of arguments to a files

classmethod write_file(*args*, *path*, *minimise=True*)
more multi-purpose writer - accepts path argument

class useful_inkleby.useful_django.views.bake.**RequestMock**(***defaults*)
Bases: django.test.client.RequestFactory

Construct a generic request object to get results of view

request(***request*)

useful_inkleby.useful_django.views.bake.**bake_static**()
syncs the static file location to the bake directory

useful_inkleby.useful_django.views.bake.**html_minify**(*x*)

useful_inkleby.useful_django.views.decorators module

useful_inkleby.useful_django.views.decorators.**use_template**(*template*)
Decorator to return a HTTPResponse from a function that just returns a dictionary.

Functions should return a dictionary.

Usage: @use_template(template_location)

useful_inkleby.useful_django.views.functional module

Created on 26 Mar 2016

@author: alex

class useful_inkleby.useful_django.views.functional.**FunctionalView**
Bases: object

Very simple class-based view that simple expects the class to have a 'template' variable and a 'view' function that expects (self, request).

Idea is to preserve cleanness of functional view logic but tidy up the most common operation.

access_denied_no_auth(*request*)
override to provide a better response if someone needs to login

access_denied_no_staff(*request*)
override to provide a better response if someone needs to be staff

classmethod as_view(*decorators=True*, *no_auth=False*)
if decorators is True - we apply any view_decorators listed for the class if no_auth = True, we bypass staff and user testing(useful for baking)

context_to_html(*request*, *context*)

extra_params(*context*)

static login_test(*u*)

require_login = False

require_staff = False

static staff_test(*u*)

template = ''

view(*request*)

view_decorators = []

class useful_inkleby.useful_django.views.functional.**LogicalView**
Bases: *useful_inkleby.useful_django.views.functional.FunctionalView*

Runs with class-based logic while trying to keep the guts exposed.

request becomes self.request

expects a 'logic' rather than a view function (no args).

giving the class an 'args' list of strings tells it what to convert view arguments into.

e.g. args = ['id_no'] - will create self.id_no from the view argument. if an arg is a tuple ('id_no','5') - will set a default value.

functions that start **prelogic_** are run before logic() functions that start **postlogic_** are run after logic()

can also use prelogic and postlogic decorators this accept an optional order argument - lower order have priority. Default order is 5.

default is 5.

args = []

logic()
any new values assigned to self will be passed to the template bare self.value becomes value

view(*request*, **args*, ***kwargs*)
override with something that returns a dictionary to use plain functional view logic

useful_inkleby.useful_django.views.functional.**handle_redirect**(*func*)

useful_inkleby.useful_django.views.mixins module

class useful_inkleby.useful_django.views.mixins.**MarkDownView**
Bases: object

allows for a basic view where a markdown files is read in and rendered

Give the class a markdown_loc variable which is the filepath to the markdown files.

use self.get_markdown() to retrieve markdown text. If using clean, it is avaliable as 'markdown' in the template.

get_markdown()

markdown_loc = ''

view(*request*)

useful_inkleby.useful_django.views.url module

IntegratedURLView - Sidestep django's url.py based setup and integrate urls directly with view classes rather than keeping them seperate.

This will mix-in either with the functional inkleby view or the default django class-based views.

In views module you set up a series of classes that inherit from IntegratedURLView and then connect up in project url like so:

url(r'^foo/', include_view('foo.views')),

Philosophy behind this is that the current urlconf system was designed for functional views - class-based views have to hide themselves as functions with an as_view function, which is ugly. By moving responsibility for generating these to the class view it avoids awkward manual repetition and keeps all settings associated with the view in one place. Apps then don't need a separate url.py.

class useful_inkleby.useful_django.views.url.**AppUrl**(*app_view*)
Bases: object

bake(***kwargs*)
bake all views with a bake_path

has_bakeable_views()

patterns()
return patterns of all associated views

class useful_inkleby.useful_django.views.url.**IntegratedURLView**
Bases: *useful_inkleby.useful_django.views.functional.FunctionalView*

Integrate URL configuration information into the View class.

Makes app level urls.py unnecessary.

add class level variables for:

url_pattern - regex string url_patterns - list of regex strings url_name - name for url view (for reverse lookup) url_extra_args - any extra arguments to be fed into the url function for this view.

classmethod get_pattern()
returns a list of conf.url objects for url patterns that match this object

classmethod redirect_response(**args*)

url_extra_args = {}

url_name = ‘’

url_pattern = ‘’

url_patterns = []

useful_inkleby.useful_django.views.url.**include_view**(*arg*, *namespace=None*, *app_name=None*)

useful_inkleby.useful_django.views.url.**make_comparison**(*v*)

Module contents

class useful_inkleby.useful_django.views.**ComboView**

Bases: *useful_inkleby.useful_django.views.bake.BakeView*, useful_inkleby.useful_django.views.social.SocialView, *useful_inkleby.useful_django.views.url.IntegratedURLView*

useful_inkleby.useful_django.fields package

Submodules

useful_inkleby.useful_django.fields.serial module

class useful_inkleby.useful_django.fields.serial.**JsonBlockField**(*verbose_name=None*, *name=None*, *primary_key=False*, *max_length=None*, *unique=False*, *blank=False*, *null=False*, *db_index=False*, *rel=None*, *default=<class django.db.models.fields.NOT_PROVIDED>*, *editable=True*, *serialize=True*, *unique_for_date=None*, *unique_for_month=None*, *unique_for_year=None*, *choices=None*, *help_text=u’‘*, *db_column=None*, *db_tablespace=None*, *auto_created=False*, *validators=[]*, *error_messages=None*)

Bases: django.db.models.fields.TextField

store a collection of generic objects in a jsonblock. Useful for when you have a hierarchy of classes that are only accessed from the one object.

from_db_value(*value*, *expression*, *connection*, *context*)

get_prep_value(*value*)

to_python(*value*)

Module contents

useful_inkleby.files package

QuickGrid Usage

```
grid = QuickGrid().open(FILEPATH)

for r in grid:
    #for each row
    print r["foo"] #print the value for the column called "foo"

for r in grid.only("city","london"):
    #for each row
    print r["foo"]

for r in grid.exclude("fruit","banana"):
    #for each row
    print r["foo"]
```

useful_inkleby.files.quickgrid module

Python Module Index

u

Index

V

W