useful_inkleby Documentation

Release 0.1

Alex Parsons

Oct 11, 2018

Contents

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

useful_inkleby.useful_django.models:

- 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:

- IntegratedURLView - Integrates django's url settings directly into the view classes rather than keeping them in a separate urls.py.
- FunctionalView - Slimmed down version of class-based views where functional logic is preserved - but class structure used to tidy up common functions.
- LogicalView - expects a logic function where any values assigned to self are passed to template. Other functions can be run before or after using @prelogic and @postlogic operators.
- SocialView - Allows liquid templates to be used to specify social share formats in class properties
- BakeView - Handles baking a view into files - expects a bake_args function that can feed it arbitrary sets of arguments and a BAKE_LOCATION in the settings.
- MarkDownView - Mixin to read a markdown file into the view.
- LogicalURLView - combines BakeView, IntegratedURLView.
- LogicalSocialView - combines BakeView, IntegratedURLView, SocialView.

`useful_inkleby.useful_django.commands`:

- bake appname - triggers baking an app
- populate appname - runs the populate.py script for an app

useful_inkleby.useful_django.fields:

- 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:

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

`useful_inkleby.decorators`:

- GenericDecorator - Cleans up creation of function decorators.

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

Move URL patterns and names into the class to get rid of app level urls.py

For the view:

```python
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:

```python
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

LogicalView Usage

Gets rid of the need for messy supers or the hidden logic of the django class views.

Expects a logic function and everything assigned to self is avaliable in view.

other functions can be run before or after with @prelogic and @postlogic

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

            @prelogic
            def run_this_before():
                    self.text_to_use = "foo"

    def logic(self):
                    self.f = self.text_to_use
```

CHAPTER 5

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

Add ‘useful_inkleby.useful_django’ to INSTALLED_APPS to use bake command.

Then you can use ‘manage.py bake appname’ to bake app.

You can finetune this by creating bake.py

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:

5.1 useful_inkleby.useful_django.models package

5.1.1 Submodules

5.1.2 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.

5.1.3 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`

`batch_id`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`batch_time`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`classmethod init_queue`()

`queue`()

Add current object to the class creation queue

`classmethod queue_length`()

`classmethod 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 save_queue_if_count`(*count=1000*)

`class` `useful_inkleby.useful_django.models.mixins.`**`StockModelHelpers`**

Bases: `object`

common functions useful for all models - diagnostics etc

5.1.4 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

5.2 useful_inkleby.useful_django.views package

5.2.1 Submodules

5.2.2 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.`**BaseBakeManager**(*views_module=None*)

Bases: `object`

Manager for bake command function Subclass as views.BakeManager to add more custom behaviour

amend_settings(***kwargs*)

bake(***kwargs*)

this is the main function

bake_app()

copy_static_files()

create_bake_dir()

get_static_destination()

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*)

5.2.3 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)

5.2.4 useful_inkleby.useful_django.views.exceptions module

Created on Aug 21, 2016

@author: Alex

`exception` `useful_inkleby.useful_django.views.exceptions.`**`RedirectException`**

Bases: `exceptions.Exception`

lets you raise a redirect from anywhere in the structure rather than requiring returns to always prioritise it

5.2.5 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.

Use prelogic and postlogic decorators to run functions before or afer logic. These accept an optional order paramter. Lower order have priority. Default order is 5.

`args = []`

`logic``()`

any new values assigned to self will be passed to the template 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*`)`

5.2.6 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*`)`

5.2.7 useful_inkleby.useful_django.views.social module

View for managing social properties of view

class `useful_inkleby.useful_django.views.social.`**`SocialView`**

Bases: `object`

Uses class properties for social arguments Allows inheritance. Template language can be used e.g.

share_title = "{{title}}"

Where a view has a 'title' variable.

`extra_params``(`*context*`)`

`page_title = ''`

`share_description = ''`

share_image = ''

share_image_alt = ''

share_site_name = ''

share_title = ''

share_twitter = ''

share_url = ''

social_settings(*context*)
run class social settings against template

twitter_share_image = ''

5.2.8 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.LogicalView*

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 (optional) 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*)

5.2.9 Module contents

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

Bases: *useful_inkleby.useful_django.views.LogicalSocialView*

Backwards compatible LogicalSocialView

class useful_inkleby.useful_django.views.**LogicalSocialView**

Bases: *useful_inkleby.useful_django.views.LogicalURLView*, *useful_inkleby.useful_django.views.social.SocialView*

Contains baking, logical structure, and integrated URl and social mix-in

class useful_inkleby.useful_django.views.**LogicalURLView**

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

Contains baking, logical structure, and integrated URl

5.3 useful_inkleby.useful_django.fields package

5.3.1 Submodules

5.3.2 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_PROVIDI 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*)

5.3.3 Module contents

5.4 useful_inkleby.files package

5.4.1 Submodules

5.4.2 useful_inkleby.files.quickgrid module

QuickGrid library - very simple communication with spreadsheets. v1

class useful_inkleby.files.quickgrid.**FlexiBook**(**args*, ***kwargs*)
Bases: `xlwt.Workbook.Workbook`

modification to xlwt library - for merging multiple sheets

add_sheet_from_ql(*ql*)

merge_qls(*qls*)

class useful_inkleby.files.quickgrid.**ItemRow**(*quick_grid_instance*, *header_dict*, **args*, ***kwargs*)
Bases: `list`

object returned while iterating through a quick list

class useful_inkleby.files.quickgrid.**QuickGrid**(*name=''*, *header=[]*, *cached=True*)
Bases: `object`

A very simple files interface - loads files into memory so basic reads can be done.

add(*row*)

add_qg(*to_merge*)

col_to_location(*col*)

combine(**args*, ***kwargs*)

count(**columns*)

counter(*col*)

drop(***kwargs*)

ensure_column(*header_item*)
if column doesn't exist, add it

exclude(*col*, *value*)
returns an generator of only rows where col != value

expand(*col*, *seperator=', '*)

generate_col(*name*, *generate_function*)

get_column(*column_id*, *unique=False*)

header_di()
returns a dictionary of 'safe' header (striped, lowered) and positions.

include(*col*, *values*)
returns an generator of only rows where value in col

iterator_source()

load_from_generator()

classmethod merge(*to_merge*)
join a bunch of QuickGrids together

only(*col*, *value*)
returns an generator of only rows where col = value

open(*filename*, *tab=''*, *header_row=0*, *force_unicode=False*, *start=0*, *limit=None*, *codec=''*, ***kwargs*)
populate from a file

remap(*new_header*)
return a new quickgrid with a reduced or changed header

rename_column (*old_name*, *new_name*)
find old header - replace with enw one

save (*filename=None*, *force_unicode=False*)
save out as a csv or xls

sort (**cols*)
sort according to columns entered (start with - to reverse)

classmethod split_csv (*filename*, *save_folder*, *column_id=0*, *unicode=False*)

classmethod split_csv_count (*filename*, *save_folder*, *limit=1000*, *force_unicode=False*)

split_on_unique (*col*, *col_no=None*)

transform (*col_name*, *function*)

unique (*col*)
returns an generator of rows with unique values in col

use_query (*query*)
use the header to populate with information out of a django query

xls_book ()
create xls book from current ql

yield_from_generator ()

`useful_inkleby.files.quickgrid.`**export_csv** (*file*, *header*, *body*, *force_unicode=False*)
exports csv (non-unicode)

`useful_inkleby.files.quickgrid.`**import_csv** (*s_file*, *unicode=False*, *start=0*, *limit=None*, *codec=”*)
imports csvs, returns head/body - switch to use different csv processor

`useful_inkleby.files.quickgrid.`**import_xls** (*s_file*, *tab=”*, *header_row=0*)
does what you’d expect

`useful_inkleby.files.quickgrid.`**import_xlsx** (*s_file*, *tab=”*, *header_row=0*)
imports xlsx files

`useful_inkleby.files.quickgrid.`**make_safe** (*v*)

5.4.3 useful_inkleby.files.quicktext module

Created on Jul 25, 2016

@author: Alex

class `useful_inkleby.files.quicktext.`**QuickText** (*text=”*, *filename=”*)
Bases: `object`

quick helper object to save as unicode text

conform_quotes ()

from_list (*ls*, *delimiter=’\r\n’*)
given a list - merge it

lines (*delimiter=’\n’*)
iterate though all lines in the file can amend with .update method

open (***kwargs*)

save (***kwargs*)

5.4.4 Module contents

Python Module Index

Index