
django-crucrudile Documentation

Release 0.9.1

Hugo Geoffroy (pstch)

July 27, 2014

1	Installation	1
1.1	From Python package index	1
1.2	From source	1
2	Getting started [TODO]	3
2.1	Examples	3
3	Reference	5
3.1	Routes and route mixins	5
3.2	Entity store and decorators	22
3.3	Entity abstract class	29
3.4	Router and router mixins	32
3.5	URL utils	41
4	Running the tests	61
4.1	Dependencies	61
4.2	Running tests using <code>setuptools</code>	61
4.3	Running tests manually	62
4.4	Code coverage	62
4.5	Call graphs	62
5	Documentation	65
6	Class structure graph	67
	Python Module Index	69

Installation

Contents

- Installation
 - From Python package index
 - From source

At this time, the only requirements to run `django-crucrudile` are **Python** (3.3, 3.4), and **Django** (1.6). Django will be installed automatically if needed.

Note : backporting for Python 2.6 or could be done easily, ping me if you need it.

1.1 From Python package index

As `django-crucrudile` is still in the initial development phase, I did not yet upload it to the Python package index. To install it, you can either install it as a Python egg with `pip`, or download the source and run `setup.py install`.

To install with `pip`:

```
pip install -e git+https://github.com/pstch/django-crucrudile.git@master#egg=django-crucrudile
```

(This installs the latest release (major, minor or patch), use `@develop` to install development version. You can also use `@tag`, replacing tag by a release name (ex: 'v1.4.1') (Git tag, see Releases tab in GitHub).

1.2 From source

To install from source

```
git clone https://github.com/pstch/django-crucrudile.git
cd django-crucrudile
python setup.py install
```

If you want the development version (default branch is `master`, containing latest release), run `git checkout develop` before `python setup.py install`

`django-crucrudile` is a Python package, and it does **not** need to be included as an application (in `INSTALLED_APPS`) in Django. You only need to import the needed modules in your Python files.

Getting started [TODO]

2.1 Examples

2.1.1 Bookstore example

Bookstore example, with three simple models.

```
>>> import tests.unit
>>> from django.db.models import Model
>>> from django_crucrudile.routers import Router, ModelRouter
>>>
>>> # needed to subclass Django Model
>>> __name__ = "tests.doctests"
>>>
>>> class Book(Model):
...     pass
>>>
>>> class Author(Model):
...     pass
>>>
>>> class Editor(Model):
...     pass

>>> router = Router(generic=True)
>>>
>>> router.register(Author, index=True)
>>> router.register(Book)
>>> router.register(Editor)

>>> print(router.get_str_tree())
...
- Router @ ^
- author-list-redirect @ ^$ RedirectView
- GenericModelRouter author @ ^author/
  - author-list-redirect @ ^$ RedirectView
  - author-delete @ ^delete$ DeleteView
  - author-update @ ^update$ UpdateView
  - author-create @ ^create$ CreateView
  - author-detail @ ^detail$ DetailView
  - author-list @ ^list$ ListView
- GenericModelRouter book @ ^book/
  - book-list-redirect @ ^$ RedirectView
```

```
- book-delete @ ^delete$ DeleteView
- book-update @ ^update$ UpdateView
- book-create @ ^create$ CreateView
- book-detail @ ^detail$ DetailView
- book-list @ ^list$ ListView
- GenericModelRouter editor @ ^editor/
- editor-list-redirect @ ^$ RedirectView
- editor-delete @ ^delete$ DeleteView
- editor-update @ ^update$ UpdateView
- editor-create @ ^create$ CreateView
- editor-detail @ ^detail$ DetailView
- editor-list @ ^list$ ListView
```


3.1 Routes and route mixins

Contents

- Routes and route mixins
 - Base route
 - Route mixins
 - * Abstract
 - Arguments
 - Parser
 - Model
 - * Concrete
 - Callback
 - View
 - Callback route
 - View route
 - Model view route

A route is an implementation of the `django_crucrudile.entities.Entity` abstract class that yields URL patterns made from its attributes. In the code, this is represented by subclassing `django_crucrudile.entities.Entity` and providing a generator in `patterns()`, yielding URL patterns made from the route attributes. When route classes provide `django_crucrudile.entities.Entity.patterns()`, it makes them become concrete implementations of the `Entity` abstract class. Route classes themselves are abstract by nature and need a definition of the abstract function `base.BaseRoute.get_callback()`.

- `CallbackRoute` : Implements `base.BaseRoute` using `mixins.callback.CallbackMixin` that provides an implementation of `base.BaseRoute.get_callback()` that returns the callback set on the route (either in `CallbackRoute.__init__()` or as class attribute)
- `ViewRoute` : Implements `base.BaseRoute` using `mixins.view.ViewMixin` that provides an implementation of `base.BaseRoute.get_callback()` that returns the a callback obtaining from the view class set on the route (either in `mixins.view.ViewMixin.__init__()` or as class attribute).
- `ModelViewRoute` : Implements `base.BaseRoute` using `mixins.view.ViewMixin` and `mixins.model.ModelMixin`, passes the model in the view keyword arguments, and can be used with Django generic views.

Note: `ModelViewRoute` Can also be used in a `django_crucrudile.routers.model.ModelRouter`

store register mapping, as it correctly uses the model given in `django_crucrudile.routers.mixins.model.ModelMixin` and `django_crucrudile.routers.mixins.model.ModelMixin.get_base_store_kwargs`, and the view class that can then be registered in the resulting router.

3.1.1 Base route

This module contains the “main” abstract route class, that provides `BaseRoute.patterns()`, yielding patterns made from the route metadata and using the callback returned by implementations of the abstract function `BaseRoute.get_callback()`.

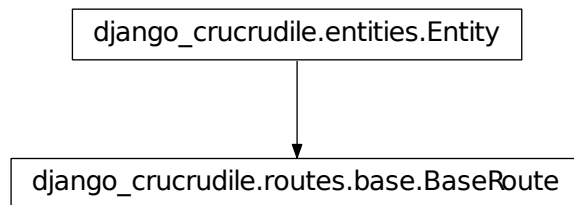
Note: This route class is one the two (along with `django_crucrudile.routers.Router`) standard implementations of the `django_crucrudile.entities.Entity` abstract class.

class `django_crucrudile.routes.base.BaseRoute` (*name=None, url_part=None, **kwargs*)
 Bases: `django_crucrudile.entities.Entity`

Abstract class for a `django_crucrudile.entities.Entity` that URL patterns that point to its implementation of `get_callback()`. Implements `django_crucrudile.entities.Entity.patterns()` using `patterns()`.

Warning: Abstract class ! Subclasses should define the `get_callback()` function. See warning in `__init__()`.

The URL part and URL name must be either set on class, or given at `__init__()`.



auto_url_part = True

Attribute auto_url_part Automatically set `url_part` to `name` if none defined.

__init__ (*name=None, url_part=None, **kwargs*)

Initialize Route, check that needed attributes/arguments are defined.

Also sets `self.redirect` to the URL name (using `get_url_name()`).

Parameters

- **name** – See `name`
- **url_part** – See `url_part`

Raises

- **ValueError** – If `name` is `None`, and not given in args

- **ValueError** – If `url_part` is `None`, and not given in `args`, and `auto_url_part` is `None` or `name` is `None`
- **TypeError** – If `get_callback()` not implemented (see warning below)

Warning: This method cannot be called on `BaseRoute`, as it is an abstract class missing an implementation of `get_callback()` :

```
>>> try:
...     BaseRoute()
... except Exception as caught:
...     type(caught).__name__
...     str(caught) == (
...         "Can't instantiate abstract class BaseRoute "
...         "with abstract methods get_callback"
...     )
'TypeError'
True
```

name = None

Attribute name URL name to use for this pattern (will be used for the Django URL pattern name)

url_part = None

Attribute url_part URL regex to use for the pattern (will be used in the URL regexp)

patterns (*parents=None, add_redirect=None, add_redirect_silent=None*)

Yield patterns for URL regexs in `get_url_regexs()`, using callback in `get_callback()` and URL name from `get_url_name()`.

Parameters

- **parents** (list of `django_crucrudile.routers.Router`) – Not used in `BaseRoute`'s implementation of `patterns`.
- **add_redirect** (*bool*) – Not used in `BaseRoute`'s implementation of `patterns`.
- **add_redirect_silent** (*bool*) – Not used in `BaseRoute`'s implementation of `patterns`.

Returns Django URL patterns

Return type iterable of `RegexURLPattern`

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = Route('name', 'url_part')
>>> list(route.patterns())
[<RegexURLPattern name ^url_part$>]
```

get_callback()

Return callback to use in the URL pattern

Warning: Abstract method ! Should be implemented in subclasses, otherwise class instantiation will fail. See warning in `__init__()`.

Returns Callable to use in the URL patter

Return type callable

get_url_name()

Get the main URL name, defined at class level (`name`) or passed to `__init__()`.

Returns main URL name

Return type str

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = Route('name', 'url_part')
>>> route.get_url_name()
'name'
```

get_url_names()

Get a list of URL names to generate patterns for. At least one URL pattern will be returned for each URL name returned by this function.

The default implementation returns a singleton (`get_url_name()`).

Returns URL names (list of URL name)

Return type iterable of str

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = Route('name', 'url_part')
>>> print(list(route.get_url_names()))
['name']
```

get_url_part()

Get the main URL part, defined at class level (`url_part`) or passed to `__init__()`.

Returns main URL part

Return type str

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = Route('name', 'url_part')
>>> route.get_url_part()
'url_part'
```

get_url_parts()

Get a list of URL parts to generate patterns for. At least one URL pattern will be returned for each URL part returned by this function.

The default implementation returns a singleton (`get_url_part()`).

Returns URL parts (list of URL part)

Return type iterable of str

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
```

```
>>>
>>> route = Route('name', 'url_part')
>>> list(route.get_url_parts())
['url_part']
```

get_url_specs()

Yield URL specifications. An URL specification is a 3-tuple, containing 3 `django_crucrudile.urlutils.URLBuilder` instances : `prefix`, `name` and `suffix`. These objects are used to join together different part of the URLs. Using them in a 3-tuple allows building an URL part with a “central” name, (whose parts are joined using `-`), a prefix (joined using `/`), and a suffix (joined using `/`, `/?`). This schema allows different subclasses to register different parts in the URLs (without interfering with each other, using `super()`), and provides automatic optional/required separator handling.

The base implementation yields a specification for each URL part returned by `get_url_parts()`.

Note: The `prefix`, `name` and `suffix` names for the URL specification contents are purely indicative, and never used as identifiers. They are used in this package’s code for consistency.

Returns URL specifications

Return type iterable of 3-tuple

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = Route('name', 'url_part')
>>> list(route.get_url_specs())
[([], ['url_part'], [])]
```

get_url_regexs()

Yield URL regexs to generate patterns for.

For each URL specification in `get_url_specs()` :

- Run each `django_crucrudile.urlutils.URLBuilder` instance in the URL specification 3-tuple (`prefix`, `name` and `suffix`)
- Pass builder outputs in another URL builder
- Run this builder, and yield output, prefixed with `^` and suffixed with `$`

URL specifications are structured as follow :

- iterable (list of
- iterable (3-tuple) of
- iterable (`URLBuilder`) of
- URL part

We use `django_crucrudile.urlutils.URLBuilder` twice, first to join the URL parts in each URL builder of the specification (`prefix`, `name` and `suffix`), and then to join together the 3 resulting URL parts.

It’s not possible to flatten this list directly, because `prefix`, `name` and `suffix` may use different separators.

Returns URL regexs

Return type iterable of string

```
>>> class Route(BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = Route('name', 'url_part')
>>> list(route.get_url_regexs())
['^url_part$']
```

3.1.2 Route mixins

This module contains route mixins, that implement specific functionality for abstract class `django_crucrudile.routes.base.BaseRoute()` subclasses. Some of these mixins make the class “concrete” (as the abstract function `django_crucrudile.routes.base.BaseRoute.get_callback()` is implemented, the class can be instantiated).

Abstract

Arguments

This module contains the `ArgumentsMixin` route mixin, that uses `parser.ArgumentsParser` to create a list of argument combinations from the given argument list.

```
class django_crucrudile.routes.mixins.arguments.ArgumentsMixin(*args, arguments_spec=None,
                                                                **kwargs)
```

Bases: `builtins.object`

Route mixin, that builds the argument combination list when instantiating, and that yields (in `get_url_specs()`) another URL specification for each argument in resulting list.

Should be a list that, if needed, contains argument specifications. An argument specification is a 2-tuple, containing a boolean indicating if the argument is required or not,

Warning: This mixin does not make `django_crucrudile.routes.base.BaseRoute` a concrete class !

`django_crucrudile.routes.mixins.arguments.ArgumentsMixin`

`arguments_parser`

Attribute `arguments_parser` Argument parser to use to build the argument combinations from the argument specifications. Should be a `django_crucrudile.urlutils.Parsable` subclass, or any class whose instances can be called to return its parsed output.

alias of `ArgumentsParser`

`__init__` (*args, arguments_spec=None, **kwargs)

Initialize route, set arguments specification if given, and run arguments parser.

Parameters `arguments_spec` – See `arguments_spec`

Example with the default test parser (`parser.ArgumentsParser`) used with `base.BaseRoute`:

```
>>> from django_crucrudile.routes.base import BaseRoute
>>>
>>> class ArgumentsRoute(ArgumentsMixin, BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> route = ArgumentsRoute(
...     'name', 'url_part',
...     arguments_spec=[
...         ['<arg1.1>', '<arg1.2>'],
...         '<arg2>'
...     ]
... )
>>>
>>> list(route.get_url_regexs())
...
['^url_part/<arg1.1>/<arg2>$',
 '^url_part/<arg1.2>/<arg2>$']
```

arguments_spec = []

Attribute `arguments_spec` Argument list that will be passed to `arguments_parser`.

Should be structured as the arguments parser expects it.

get_url_specs ()

Yield another URL specification for each argument in the argument combination list (arguments parser output).

Returns URL specifications

Return type iterable of 3-tuple

Example using the default test parser (`parser.ArgumentsParser`):

```
>>> from django_crucrudile.routes.base import BaseRoute
>>>
>>> class ArgumentsRoute(ArgumentsMixin, BaseRoute):
...     def get_callback(self):
...         pass
...     arguments_spec=[
...         ['<arg1.1>', '<arg1.2>'],
...         '<arg2>'
...     ]
>>>
>>> route = ArgumentsRoute(
...     'name', 'url_part',
... )
>>>
>>> list(route.get_url_specs())
...
[([],
 ['url_part'],
 [(True, '<arg1.1>/<arg2>')]),
 ([],
```

```
['url_part'],
[(True, '<arg1.2>/<arg2>')]]]
```

Parser This module contains the default arguments parser (`ArgumentsParser`), used in `django_crucrudile.routes.mixins.arguments.ArgumentsMixin`.

The `combine()` function is used by the cartesian product in `ArgumentsParser.cartesian_product()`, it joins an iterable (filtering out its items that evaluate to `None`) using a given separator.

`django_crucrudile.routes.mixins.arguments.parser.combine(iterable, separator)`
Join iterable (filtering out its items that evaluate to one) using separator

Parameters

- **iterable** (*iterable*) – Iterable to filter and join
- **separator** (*str*) – Separator to join with

Returns Joined string

Return type `str`

```
>>> combine(['Foo', '', 'Bar', None, 'Xyz', 0], '/')
'Foo/Bar/Xyz'
```

class `django_crucrudile.routes.mixins.arguments.parser.ArgumentsParser` (*iterable=None, separator=None, opt_separator=None, required_default=None*)

Bases: `django_crucrudile.urlutils.OptionalPartList`

This parser reads a list of argument specification, and builds an argument combination list (using a cartesian product). It subclasses `django_crucrudile.urlutils.OptionalPartList` (as an arguments list is an URL part list), and add its building parsers in `ArgumentsParser.get_parsers()`.

The input of the parser should be a list of argument specifications. Argument specifications can be written as :

- `(bool, string)` : converted to `(bool, list([string]))`
- `string` : converted to `(True, list([string]))`
- `list` : converted to `(True, list)`

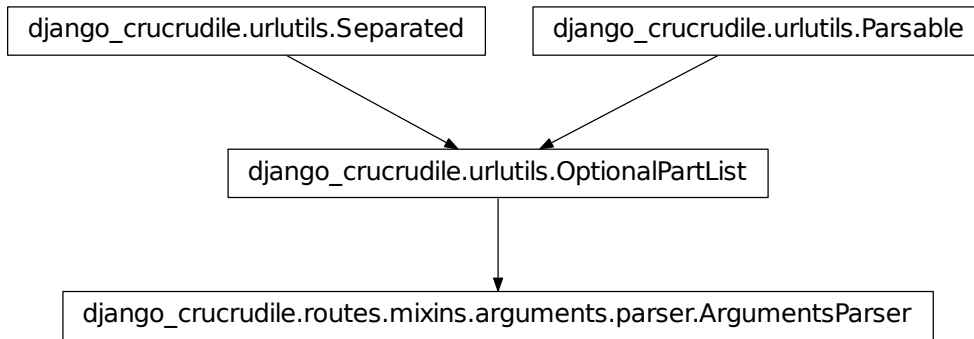
If `bool` is not defined, a default value will be used (see `django_crucrudile.urlutils.Separated.required_default`)

In `(bool, list)` :

- `bool` is a boolean flag indicating whether an argument list is required
- `list` is a list of argument, as “choices” : a combination will be generated for each item in the list

The output of the parser is a list of 2-tuple containing a boolean value and a string. The boolean value is a flag indicating whether the first argument of the string is required, and the string is the joined URL parts of the argument combination.

To set the separators, see `django_crucrudile.urlutils.Separated.separator` and `django_crucrudile.urlutils.Separated.opt_separator`.



Test, with first argument **required** :

```

>>> parser = ArgumentsParser([
...     ["<arg1.1>", "<arg2.2>"],
...     "<arg3>",
...     (False, ["<arg4.1>", "<arg4.2>"]),
...     (True, ["<args5>"])
... ])
>>>
>>> list(parser())
...
[(True, '<arg1.1>/<arg3>/?<arg4.1>/<args5>'),
 (True, '<arg1.1>/<arg3>/?<arg4.2>/<args5>'),
 (True, '<arg2.2>/<arg3>/?<arg4.1>/<args5>'),
 (True, '<arg2.2>/<arg3>/?<arg4.2>/<args5>')]
  
```

Test, with first argument **not required** :

```

>>> parser = ArgumentsParser([
...     (False, ["<arg1.1>", "<arg2.2>"]),
...     "<arg3>",
...     (False, ["<arg4.1>", "<arg4.2>"]),
...     (True, ["<args5>"])
... ])
>>>
>>> list(parser())
...
[(False, '<arg1.1>/<arg3>/?<arg4.1>/<args5>'),
 (False, '<arg1.1>/<arg3>/?<arg4.2>/<args5>'),
 (False, '<arg2.2>/<arg3>/?<arg4.1>/<args5>'),
 (False, '<arg2.2>/<arg3>/?<arg4.2>/<args5>')]
  
```

get_parsers()

Add `transform_args_to_list()`, `cartesian_product()` and `consume_cartesian_product()` to the parsers from `django_crucrudile.urlutils.OptionalPartList`

Returns Argument parsers list

Return type list of callable

static transform_args_to_list(items)

Transform second part of each item in items in a list if it's not one.

Parameters `items` (*iterable of 2-tuple*) – List of items to transform

Returns Transformed list

Return type iterable of 2-tuple : [(bool, list)]

```
>>> list(ArgumentsParser.transform_args_to_list([
...     (None, '<arg1>'),
...     (None, ['<arg2>', '<arg3>'])
... ]))
...
[(None, ['<arg1>']),
 (None, ['<arg2>', '<arg3>'])]
```

static `cartesian_product` (`items`, `get_separator`)

Process cartesian product to get all possible combinations with argument lists in `items`.

Parameters `items` (*iterable of 2-tuple*) – List of tuple to transform (2-tuple with a flag indicating if the argument specification is required, and the argument choice list)

Returns List of 2-tuple, with a flag indicating if the first item is required, and the joined list.

Return type iterable of 2-tuple : [(bool, str)]

```
>>> get_separator = lambda x: '/' if x else '/?'
```

With first spec **required** :

```
>>> list(ArgumentsParser.cartesian_product(
...     [
...         (True, ['<arg1>']),
...         (True, ['<arg2>', '<arg3>']),
...         (False, ['<arg4>', '<arg5>'])
...     ],
...     get_separator=get_separator
... ))
...
[(True, '<arg1>/<arg2>/?<arg4>'),
 (True, '<arg1>/<arg2>/?<arg5>'),
 (True, '<arg1>/<arg3>/?<arg4>'),
 (True, '<arg1>/<arg3>/?<arg5>')]
```

With first spec **not required** :

```
>>> list(ArgumentsParser.cartesian_product(
...     [
...         (False, ['<arg1>']),
...         (True, ['<arg2>', '<arg3>']),
...         (False, ['<arg4>', '<arg5>'])
...     ],
...     get_separator=get_separator
... ))
...
[(False, '<arg1>/<arg2>/?<arg4>'),
 (False, '<arg1>/<arg2>/?<arg5>'),
 (False, '<arg1>/<arg3>/?<arg4>'),
 (False, '<arg1>/<arg3>/?<arg5>')]
```

static `consume_cartesian_product` (`items`)

Force the generated to be consumed

Parameters `items` (*iterable*) – Generator to consume

Returns Consumed list

Return type list

```
>>> ArgumentsParser.consume_cartesian_product(iter([1]))
[1]
```

Model

This module contains `ModelMixin`, a route mixin that can be used to bind a model to a route, and use it when computing route metadata.

```
class django_crucrudile.routes.mixins.model.ModelMixin(*args, model=None,
**kwargs)
```

Bases: `builtins.object`

Route mixin that requires a model to be set either on the class (`model` attribute), or to be passed in `__init__()`, and provides the URL name and URL part using the model metadata.

Warning: This mixin does not make `django_crucrudile.routes.base.BaseRoute` a concrete class !

`django_crucrudile.routes.mixins.model.ModelMixin`

prefix_url_part = False

Attribute prefix_url_part Prefix the URL part with the model (ex: `/model/<url_part>`)

__init__ (*args, model=None, **kwargs)

Initialize ModelRoute, check that model is defined at class-level or passed as argument.

Parameters `model` – See `model`

Raises ValueError `model` argument is None, and no model defined in `model`

model = None

Attribute model Model to use on the Route

model_url_name

Return the model name to be used when building the URL name

Returns URL name from model name, using Django internals

Return type str

```
>>> from django_crucrudile.routes.base import BaseRoute
>>> from mock import Mock
>>>
>>> class ModelRoute(ModelMixin, BaseRoute):
...     def get_callback(self):
```

```
...     pass
>>>
>>> model = Mock()
>>> model._meta.model_name = 'testmodel'
>>> route = ModelRoute(model=model, name='routename')
>>>
>>> route.model_url_name
'testmodel'
```

model_url_part

Return the model name to be used when building the URL part

Returns URL part from the URL name (`model_url_name()`)

Return type str

```
>>> from django_crucrudile.routes.base import BaseRoute
>>> from mock import Mock
>>>
>>> model = Mock()
>>> model._meta.model_name = 'testmodel'
>>> route = ModelMixin(model=model)
>>>
>>> route.model_url_part
'testmodel'
```

get_url_specs()

Return URL specs where the model URL name is appended to the prefix part list if needed

Returns URL specifications

Return type iterable of 3-tuple

```
>>> from django_crucrudile.routes.base import BaseRoute
>>> from mock import Mock
>>>
>>> class ModelRoute(ModelMixin, BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> model = Mock()
>>> model._meta.model_name = 'testmodel'
>>> route = ModelRoute(model=model, name='routename')
>>>
>>> list(route.get_url_specs())
[([], ['routename'], [])]
```

With `prefix_url_part` set to True:

```
>>> from django_crucrudile.routes.base import BaseRoute
>>> from mock import Mock
>>>
>>> class PrefixModelRoute(ModelMixin, BaseRoute):
...     def get_callback(self):
...         pass
...     prefix_url_part = True
>>>
>>> model = Mock()
>>> model._meta.model_name = 'testmodel'
>>> route = PrefixModelRoute(
...     model=model, name='routename',
```

```
... )
>>>
>>> list(route.get_url_specs())
...
[(['testmodel'],
  ['routename'],
  [])]
```

`get_url_name()`

Return the URL name built `model_url_name()` and `Route.name`.

Returns compiled URL name

Return type str

```
>>> from django_crucrudile.routes.base import BaseRoute
>>> from mock import Mock
>>>
>>> class ModelRoute(ModelMixin, BaseRoute):
...     def get_callback(self):
...         pass
>>>
>>> model = Mock()
>>> model._meta.model_name = 'testmodel'
>>> route = ModelRoute(
...     model=model, name='routename',
... )
>>>
>>> route.get_url_name()
'testmodel-routename'
```

Concrete

Callback

This module contains a route mixin, `CallbackMixin`, that implements `django_crucrudile.routes.base.BaseRoute`.

class `django_crucrudile.routes.mixins.callback.CallbackMixin` (*args, callback=None, **kwargs)

Bases: `builtins.object`

Route mixin, implements `django_crucrudile.routes.base.BaseRoute`, requires a callback to be set either on the class (`callback` attribute), or to be passed in `__init__()`.

Note: This mixin makes the class concrete, as it implements the `django_crucrudile.routes.base.BaseRoute.get_callback()` abstract function.

`django_crucrudile.routes.mixins.callback.CallbackMixin`

`__init__ (*args, callback=None, **kwargs)`

Initialize CallbackRoute, check that callback is defined at class-level or passed as argument

Parameters `callback` – See [callback](#)

Raises `ValueError` If `callback` and `callback` are both None

`callback = None`

Attribute `callback` Callback that will be used by the URL pattern

`get_callback ()`

Return `callback`

Returns The callback set on class (`callback`) or passed to `__init__ ()`.

Return type callable

View

This module contains a route mixin, `ViewMixin`, that implements `django_crucrudile.routes.base.BaseRoute`.

```
class django_crucrudile.routes.mixins.view.ViewMixin (view_class=None, name=None,
                                                    auto_url_name_from_view=None,
                                                    *args, **kwargs)
```

Bases: `builtins.object`

Route mixin, implements `django_crucrudile.routes.base.BaseRoute`, requires a view class to be set either on the class (`callback` attribute), or to be passed in `__init__ ()`.

The view class will be used to get the callback to give to the URL pattern.

Note: This mixin makes the class concrete, as it implements the `django_crucrudile.routes.base.BaseRoute.get_callback ()` abstract function.

`django_crucrudile.routes.mixins.view.ViewMixin`

```
>>> class TestView:
...     pass
>>>
>>> route = ViewMixin(TestView)
>>>
>>> route.view_class.__name__
'TestView'
>>> route.name
'test'
```

With `auto_url_name_from_view` set to False :

```
>>> class TestView:
...     pass
```

```
>>>
>>> route = ViewMixin(TestView, auto_url_name_from_view=False)
>>>
>>> route.view_class.__name__
'TestView'
>>>
>>> # Because :class: 'ViewMixin' does not even set
>>> # :attr: 'django_crucrudile.routes.base.BaseRoute.name' if
>>> # :attr: 'auto_url_name_from_view' is 'False', this will
>>> # raise an attribute error :
>>> route.name
Traceback (most recent call last):
...
AttributeError: 'ViewMixin' object has no attribute 'name'
```

__init__ (view_class=None, name=None, auto_url_name_from_view=None, *args, **kwargs)
Initialize ViewRoute, check that view_class is defined at class-level or passed as argument.

Parameters

- **view_class** – See [view_class](#)
- **auto_url_name_from_view** – See [auto_url_name_from_view](#)

See also:

For doctests that use this member, see [ViewMixin](#)

Raises ValueError if both `view_class` and `view_class` are None

view_class = None

Attribute view_class View class that will be used to get the callback to pass to the URL pattern

auto_url_name_from_view = True

Attribute auto_url_name_from_view Automatically set route name using view class name (lower casing it, and stripping it of View)

get_callback()

Return callback using `django.generic.views.View.as_view()`, getting arguments from `get_view_kwargs()`.

Calls `View.as_view()` on view class, with kwargs from `get_view_kwargs()`, to get callback to use in URL pattern.

Returns Callback to use in URL pattern

Return type callable

```
>>> from mock import Mock
>>> callback = Mock()
>>> mock_view = Mock()
>>> mock_view.__name__ = 'MockView'
>>> mock_view.as_view = lambda: callback
>>>
>>> route = ViewMixin(view_class=mock_view)
>>> route.get_callback() is callback
True
```

get_view_kwargs()

Return arguments to use when calling the callback builder.

Returns Keyword arguments

Return type dict

classmethod `make_for_view(view_class, **kwargs)`

Return a subclass of this class, setting the `view_class` argument at class-level.

Also sets the `django_crucrudile.routes.base.BaseRoute.name` attribute using the view class name.

This is useful when combined with `django_crucrudile.entities.store.EntityStore.register_class` as it only accepts classes (in opposition to `django_crucrudile.entities.store.EntityStore.register()`)

Parameters `view_class` (subclass of `django.views.generic.view`) – View class to set on the resulting class

Returns New class, with `view_class` attribute set to `view_class` argument.

```
>>> class TestView:
...     pass
>>>
>>> route_class = ViewMixin.make_for_view(TestView)
>>>
>>> route_class.__name__
'TestRoute'
>>> route_class.view_class.__name__
'TestView'
```

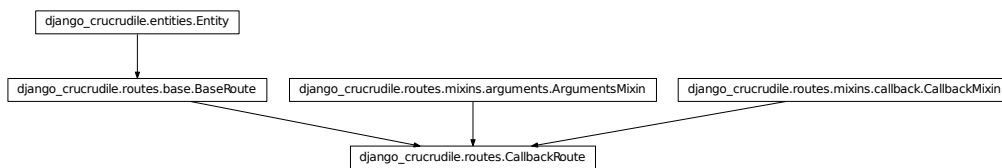
3.1.3 Callback route

class `django_crucrudile.routes.CallbackRoute(*args, **kwargs)`

Bases: `django_crucrudile.routes.mixins.arguments.ArgumentsMixin`,
`django_crucrudile.routes.mixins.callback.CallbackMixin`,
`django_crucrudile.routes.base.BaseRoute`

Implement `base.BaseRoute` using a callback function.

Also use `mixins.arguments.ArgumentsMixin` to allow URL arguments to be specified.



__init__ `(*args, **kwargs)`

Initialize `CallbackRoute`, for a description of arguments see :

- `mixins.arguments.ArgumentsMixin.__init__()`
- `mixins.callback.CallbackMixin.__init__()`
- `base.BaseRoute.__init__()`

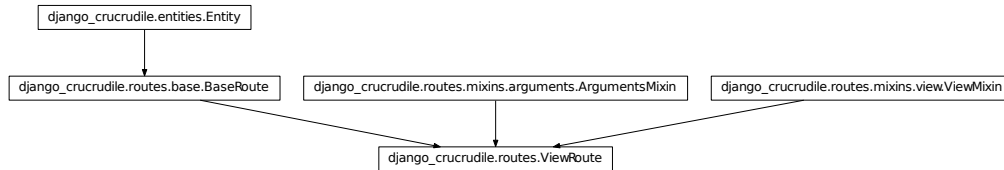
3.1.4 View route

class `django_crucrudile.routes.ViewRoute(*args, **kwargs)`

Bases: `django_crucrudile.routes.mixins.arguments.ArgumentsMixin`,
`django_crucrudile.routes.mixins.view.ViewMixin`, `django_crucrudile.routes.base.BaseRoute`

Implement `base.BaseRoute` using a view class function.

Also use `mixins.arguments.ArgumentsMixin` to allow URL arguments to be specified.



__init__ (*args, **kwargs)

Initialize ViewRoute, for a description of arguments see :

- `mixins.arguments.ArgumentsMixin.__init__()`
- `mixins.view.ViewMixin.__init__()`
- `base.BaseRoute.__init__()`

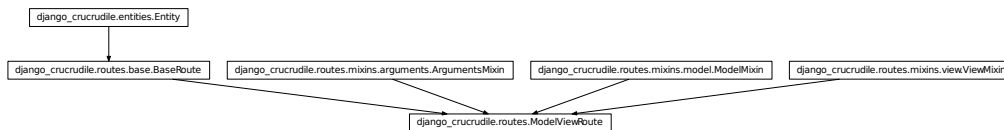
3.1.5 Model view route

class `django_crucrudile.routes.ModelViewRoute(*args, **kwargs)`

Bases: `django_crucrudile.routes.mixins.arguments.ArgumentsMixin`,
`django_crucrudile.routes.mixins.model.ModelMixin`, `django_crucrudile.routes.mixins.view.ViewMixin`,
`django_crucrudile.routes.base.BaseRoute`

Combine `mixins.view.ViewMixin` and `django_crucrudile.routes.mixins.model.ModelMixin` to make a route that can easily be used with a model and a generic view.

Also use `mixins.arguments.ArgumentsMixin` to allow URL arguments to be specified.



__init__ (*args, **kwargs)

Initialize ModelViewRoute, for a description of arguments see :

- `mixins.arguments.ArgumentsMixin.__init__()`
- `mixins.model.ModelMixin.__init__()`

```
•mixins.view.ViewMixin.__init__()
•base.BaseRoute.__init__()

get_view_kwargs()
    Make the view use mixins.model.ModelMixin.model.

    This is the effective combination of mixins.model.ModelMixin and ViewRoute.

>>> from mock import Mock
>>>
>>> model = Mock()
>>> route = ModelViewRoute(model=model, view_class=Mock(), name='name')
>>>
>>> route.get_view_kwargs()['model'] is model
True
```

3.2 Entity store and decorators

Contents

- Entity store and decorators
 - Decorators
 - Entity store

The entity store class provides functions to register entity instances in an entity store.

It also allow entity classes to be registered at class-level in a “base store”. These entity classes will be instantiated when the entity store gets instantiated, and the resulting instances will be registered.

It also allows “register mappings” to be defined, they allow objects passed to register functions to be transformed based on a mapping. When looking for a mapping, register functions will compare their argument to the mapping key (using `issubclass`, or `isinstance`), and use the corresponding mapping value to get the object that will be registered.

Additionally, register mapping keys can be iterables (list or tuples), in which case the register functions will compare their argument to any of the elements in the mapping key, and match even if only a single item in the mapping key matches the argument.

The base store is copied on each class definition, using a metaclass, so that using register functions that class-level won’t alter the base store of other class definitions.

This module also contains a `provides()` decorator, that decorates a entity store class, adding an object to its base store.

Doctests that use functionality in `EntityStore` can be seen in other classes (in particular `django_crucrudile.routers.Router`). They may help to get a good idea of what the entity, entity store and entity graph concepts mean.

3.2.1 Decorators

`django_crucrudile.entities.store.provides(provided, **kwargs)`

Return a decorator that uses `EntityStore.register_class()` to register the given object in the base store.

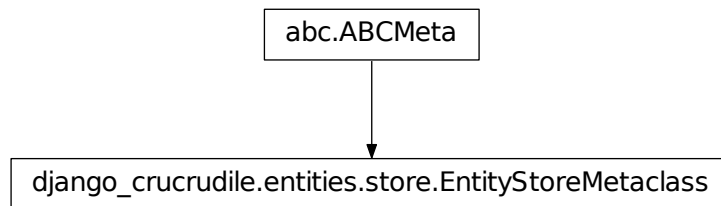
Parameters `provided` (*object*) – Class (or object) to register in the base store. This can be an object since it may be transformed by `:func:'EntityStore.register_apply_map'd`

3.2.2 Entity store

class `django_crucrudile.entities.store.EntityStoreMetaclass` (*name, bases, attrs*)
 Bases: `abc.ABCMeta`

`EntityStoreMetaclass` allows `EntityStore` to use a different `_base_store` store (list instance) for each class definitions (`cls` instantiation)

Note: Subclasses `abc.ABCMeta` because it will be used as the metaclass for an entity, and entity are abstract classes, which needs the `abc.ABCMeta` base class.



```

>>> class Store(metaclass=EntityStoreMetaclass):
...     pass
>>>
>>> class FailStore:
...     _fail_store = []
>>>
>>> class NewStore(Store):
...     pass
>>>
>>> class FailNewStore(FailStore):
...     pass

>>> (NewStore._base_store is
...   Store._base_store)
False
>>> (NewStore._base_register_map is
...   Store._base_register_map)
False
>>> (NewStore._base_register_class_map is
...   Store._base_register_class_map)
False

>>> (FailNewStore._fail_store is
...   FailStore._fail_store)
True
  
```

class `django_crucrudile.entities.store.EntityStore`
 Bases: `builtins.object`

Provides an entity store, and a `register()` method that registers entities in the entity store.

The subclass implementation of `patterns()` should iterate over the entity store.

django_crucrudile.entities.store.EntityStore

static register_apply_map (*entity, mapping, transform_kwargs=None, silent=True*)

Apply mapping of value in mapping if *entity* is subclass (`issubclass()`) or instance (`isinstance()`) of key

Parameters

- **entity** (*object or class*) – Object to pass to found mappings
- **mapping** (*dict*) – Register mapping, used to get callable to pass *entity* to
- **transform_kwargs** (*dict*) – Extra keyword arguments to pass to the found transform functions (mapping keys)
- **silent** – If set to `False`, will fail if not matching mapping was found.

Raises LookupError If *silent* is `False`, and no matching mapping was found

```
>>> from mock import Mock
>>>
>>> class Class:
...     pass
>>> class SubClass(Class):
...     pass
>>> class OtherClass:
...     pass
>>>
>>> instance = SubClass()
```

With instance :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     instance,
...     {Class: class_mock}
... )
>>> class_mock.assert_called_once_with(instance)
```

With instance, and default mapping :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     instance,
...     {None: class_mock}
... )
>>> class_mock.assert_called_once_with(instance)
```

With instance and iterable bases :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     instance,
...     {(OtherClass, Class): class_mock}
```

```
... )
>>> class_mock.assert_called_once_with(instance)
```

With instance and iterable bases (no matching base) :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     instance,
...     {(OtherClass, ): class_mock}
... )
>>> applied is instance
True
>>> class_mock.called
False
```

With instance and iterable bases (no matching base, not silent) :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     instance,
...     {(OtherClass, ): class_mock},
...     silent=False
... )
...
Traceback (most recent call last):
...
LookupError: Could not find matching key in register
mapping. Used test 'isinstance', register mapping bases are
'OtherClass', tested against 'SubClass'
```

With subclass :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     SubClass,
...     {Class: class_mock}
... )
>>> class_mock.assert_called_once_with(SubClass)
```

With subclass and iterable bases (no matching base) :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     SubClass,
...     {(OtherClass, ): class_mock}
... )
>>> applied is SubClass
True
>>> class_mock.called
False
```

With subclass and single bases (no matching base, not silent) :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     SubClass,
...     {OtherClass: class_mock},
...     silent=False
... )
...
```

```
Traceback (most recent call last):
...
LookupError: Could not find matching key in register
mapping. Used test 'issubclass', register mapping bases are
'OtherClass', tested against 'SubClass'
```

With subclass and no mappings (not silent) :

```
>>> class_mock = Mock()
>>> applied = EntityStore.register_apply_map(
...     SubClass,
...     {},
...     silent=False
... )
...
Traceback (most recent call last):
...
LookupError: Could not find matching key in register
mapping. Used test 'issubclass', register mapping bases are
'', tested against 'SubClass'
```

classmethod `get_register_class_map()`

Mapping of type to function that will be evaluated (with entity) when calling register. See `register_class()` and `register_apply_map()`.

Overriding implementations must call the base implementation (using `super()`, usually), so that the base mappings set by `set_register_class_mapping()` can be returned.

The base implementation returns a copy of the stored mapping, so overriding implementations may append to the return value.

See also:

For doctests that use this member, see `django_crucrudile.entities.store.EntityStore.register_class_map()`.

classmethod `get_register_class_map_kwargs()`

Arguments passed when applying register map, in `register_class()`

See also:

For doctests that use this member, see `django_crucrudile.entities.store.EntityStore.register_class_map_kwargs()`.

`get_register_map()`

Mapping of type to function that will be evaluated (with entity) when calling register. See `register()` and `register_apply_map()`

Overriding implementations *MUST* call the base implementation (using `super()`, usually), so that the base mappings set by `set_register_mapping()` can be returned.

The base implementation returns a copy of the stored mapping, so overriding implementations may append to the return value.

See also:

For doctests that use this member, see `django_crucrudile.entities.store.EntityStore.register_map()`.

`get_register_map_kwargs()`

Arguments passed when applying register map, in `register()`

See also:

For doctests that use this member, see `django_crucrudile.entities.store.EntityStore.register_map_kwargs()`.

classmethod `set_register_class_mapping` (*key*, *value*)

Set a base register class mapping, that will be returned (possibly with other mappings) by `get_register_class_map()`.

Parameters

- **key** (*class or tuple of classes*) – Register class mapping bases
- **value** (*callable*) – Register class mapping value

```
>>> from mock import Mock
>>> mock_mapping_func = Mock()
>>>
>>> class Class:
...     pass
>>> class Store(EntityStore):
...     pass
>>>
>>> Store.set_register_class_mapping(
...     Class, mock_mapping_func
... )
>>> Store.get_register_class_map() == (
...     {Class: mock_mapping_func}
... )
True
```

classmethod `set_register_mapping` (*key*, *value*)

Set a base register mapping, that will be returned (possibly with other mappings) by `get_register_map()`.

Parameters

- **key** (*class or tuple of classes*) – Register mapping bases
- **value** (*callable*) – Register mapping value

```
>>> from mock import Mock
>>> mock_mapping_func = Mock()
>>>
>>> class Class:
...     pass
>>>
>>> class Store(EntityStore):
...     pass
>>>
>>> Store.set_register_mapping(
...     Class, mock_mapping_func
... )
>>> Store().get_register_map() == (
...     {Class: mock_mapping_func}
... )
True
```

classmethod `register_class` (*register_cls*, *map_kwargs=None*)

Add a route class to `_base_store`, applying mapping from `get_register_class_map()` where required. This route class will be instantiated (with kwargs from `get_base_store_kwargs()`) when the Router is itself initiated, using `register_base_store()`.

Parameters

- **register_cls** – Object to register (usually Route or Router classes, but could be anything because of possible mapping in `get_register_class_map_kwargs()`)
- **map_kwargs** (*dict*) – Argument to pass to mapping value if entity gets transformed.

Returns The registered class, transformed by class register mappings if there was a matching mapping

Return type `class`

```
>>> from mock import Mock
>>> mock_entity_instance = Mock()
>>> mock_entity = Mock()
>>> mock_entity.side_effect = [mock_entity_instance]
>>> mock_mapping_func = Mock()
>>> mock_mapping_func.side_effect = [mock_entity]
>>>
>>> class Class:
...     pass
>>>
>>> class Store(EntityStore):
...     @classmethod
...     def get_register_class_map(self):
...         return {Class: mock_mapping_func}
>>>
>>> Store.register_class(Class) is mock_entity
True
>>>
>>> Store._base_store == [mock_entity]
True
>>>
>>> store = Store()
>>> store._store == [mock_entity_instance]
True
```

register (*entity*, *map_kwargs=None*)

Register routed entity, applying mapping from `get_register_map()` where required

Parameters

- **entity** (`django_crucrudile.entities.Entity`) – Entity to register
- **map_kwargs** (*dict*) – Argument to pass to mapping value if entity gets transformed.

Returns The registered entity, transformed by register mappings if there was a matching mapping

Return type `django_crucrudile.entities.Entity`

```
>>> from mock import Mock
>>> mock_entity_instance = Mock()
>>> mock_mapping_func = Mock()
>>> mock_mapping_func.side_effect = [mock_entity_instance]
>>>
>>> class Class:
...     pass
>>> instance = Class()
>>>
>>> class Store(EntityStore):
...     @classmethod
...     def get_register_map(self):
...         return {Class: mock_mapping_func}
>>>
```



```
>>>
>>> store = Store()
>>> store.register(instance) == mock_entity_instance
True
>>> store._store == [mock_entity_instance]
True
```

`get_base_store_kwargs()`

Arguments passed when instantiating entity classes in `_base_store`

Returns Keyword arguments

Return type dict

```
>>> from mock import Mock
>>> mock_entity = Mock()
>>>
>>> class Store(EntityStore):
...     def get_base_store_kwargs(self):
...         return {'x': mock_entity}
>>>
>>> Store.register_class(lambda x: x) is not None
True
>>>
>>> store = Store()
>>> store._store == [mock_entity]
True
```

`register_base_store()`

Instantiate entity classes in `_base_store`, using arguments from `get_base_store_kwargs()`

```
>>> class Store(EntityStore):
...     pass
>>>
>>> Store.register_class(lambda: None) is not None
True
>>>
>>> store = Store()
>>> store._store
[None]
```

3.3 Entity abstract class

Contents

- Entity abstract class

An entity is an abstract class of objects that can be used to make an URL pattern tree.

See also:

In `django-crucrudile`, there are two classes that directly subclass `Entity`:

- `django_crucrudile.routers.Router` (concrete)
- `django_crucrudile.routes.base.BaseRoute` (abstract)

```
class django_crucrudile.entities.Entity(index=None)
    Bases: builtins.object
```

An entity is an abstract class of objects that can be used to make an URL pattern tree.

Abstract class that defines an attribute (`Entity.index`), and an abstract method (`Entity.patterns()`). Entity implementations should provide the `Entity.patterns()` method, that should return a generator yielding Django URL patterns.

Warning: Abstract class ! Subclasses should define the abstract `patterns()` method, that should return a generator yielding Django URL objects (`django.core.urlresolvers.RegexURLPattern` or `:class:'django.core.urlresolvers.RegexURLResolver`). See warning in `__init__()`.

`django_crucrudile.entities.Entity`

index = False

Attribute index Used when routed entity is registered, to know if it should be registered as index.

patterns (*parents=None, add_redirect=None, add_redirect_silent=None*)
Yield URL patterns

Note: For argument specification, see implementations of this abstract function (in particular `django_crucrudile.routers.Router.patterns()`)

Warning: Abstract method ! Should be defined by subclasses, and should return a generator yielding Django URL objects (`RegexURLPattern` or `RegexURLResolver`)

get_str_tree (*patterns_kwargs=None, indent_char=' ', indent_size=2*)
Return the representation of a entity patterns structure

Parameters

- **patterns_kwargs** (*dict*) – Keyword arguments to pass to `patterns()`
- **indent_char** (*str*) – String to use for tree indentation
- **indent_size** (*int*) – Indent size

```
>>> import tests.unit
>>> from django.db.models import Model
>>> from django_crucrudile.routers import Router, ModelRouter
>>>
>>> # needed to subclass Django Model
>>> __name__ = "tests.doctests"
>>>
>>> class TestModel(Model):
...     pass
```

```
>>> router = Router(generic=True)
>>>
>>> router.register(TestModel)

>>> print(router.get_str_tree())
...
- Router @ ^
  - GenericModelRouter testmodel @ ^testmodel/
    - testmodel-list-redirect @ ^$ RedirectView
    - testmodel-delete @ ^delete$ DeleteView
    - testmodel-update @ ^update$ UpdateView
    - testmodel-create @ ^create$ CreateView
    - testmodel-detail @ ^detail$ DetailView
    - testmodel-list @ ^list$ ListView
```

class `django_crucrudile.entities.Entity` (*index=None*)

Bases: `builtins.object`

An entity is an abstract class of objects that can be used to make an URL pattern tree.

Abstract class that defines an attribute (`Entity.index`), and an abstract method (`Entity.patterns()`). Entity implementations should provide the `Entity.patterns()` method, that should return a generator yielding Django URL patterns.

Warning: Abstract class ! Subclasses should define the abstract `patterns()` method, that should return a generator yielding Django URL objects (`django.core.urlresolvers.RegexURLPattern` or `:class:'django.core.urlresolvers.RegexURLResolver`). See warning in `__init__()`.

`django_crucrudile.entities.Entity`

index = False

Attribute index Used when routed entity is registered, to know if it should be registered as index.

patterns (*parents=None, add_redirect=None, add_redirect_silent=None*)

Yield URL patterns

Note: For argument specification, see implementations of this abstract function (in particular `django_crucrudile.routers.Router.patterns()`)

Warning: Abstract method ! Should be defined by subclasses, and should return a generator yielding Django URL objects (`RegexURLPattern` or `RegexURLResolver`)

get_str_tree (*patterns_kwargs=None, indent_char=' ', indent_size=2*)

Return the representation of a entity patterns structure

Parameters

- `patterns_kwargs` (*dict*) – Keyword arguments to pass to `patterns()`
- `indent_char` (*str*) – String to use for tree indentation
- `indent_size` (*int*) – Indent size

```
>>> import tests.unit
>>> from django.db.models import Model
>>> from django_crucrudile.routers import Router, ModelRouter
>>>
>>> # needed to subclass Django Model
>>> __name__ = "tests.doctests"
>>>
>>> class TestModel(Model):
...     pass

>>> router = Router(generic=True)
>>>
>>> router.register(TestModel)

>>> print(router.get_str_tree())
...
- Router @ ^
- GenericModelRouter testmodel @ ^testmodel/
- testmodel-list-redirect @ ^$ RedirectView
- testmodel-delete @ ^delete$ DeleteView
- testmodel-update @ ^update$ UpdateView
- testmodel-create @ ^create$ CreateView
- testmodel-detail @ ^detail$ DetailView
- testmodel-list @ ^list$ ListView
```

3.4 Router and router mixins

Contents

- Router and router mixins
 - Base router
 - Router mixins
 - * Model router mixin
 - Model router
 - * Generic model router

A router is an implementation of the abstract class `Entity`, that uses an entity store to allow routing other entities. In the code, this is represented by subclassing `django_crucrudile.entities.store.EntityStore` and `django_crucrudile.entities.Entity`, and providing a generator in `patterns()`, yielding URL patterns made from the entity store. Providing `django_crucrudile.entities.Entity.patterns()` makes router classes concrete implementations of the `Entity` abstract class, which allows them to be used in entity stores.

This module contains three implementations of routers, a simple one, and two implementations adapted to Django models :

- `Router` : implements the abstract class `django_crucrudile.entities.Entity`, and subclassing `django_crucrudile.entities.store.EntityStore` to implement `Router.patterns()`
- `model.ModelRouter` : subclasses `Router`, instantiate with a model as argument, adapted to pass that model as argument to registered entity classes

- `model.generic.GenericModelRouter` : that subclasses `model.ModelRouter` along with a set of default `django_crucrudile.routes.ModelViewRoute` for the five default Django generic views.

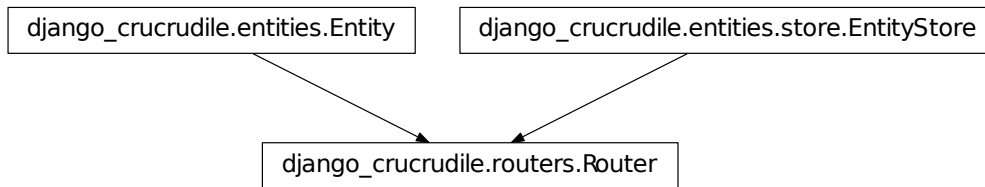
3.4.1 Base router

class `django_crucrudile.routers.Router` (`namespace=None`, `url_part=None`, `redirect=None`, `add_redirect=None`, `add_redirect_silent=None`, `get_redirect_silent=None`, `generic=None`, `**kwargs`)

Bases: `django_crucrudile.entities.store.EntityStore`, `django_crucrudile.entities.Entity`

RoutedEntity that yields an URL group containing URL patterns from the entities in the entity store (`django_crucrudile.entities.store.EntityStore`). The URL group can be set have an URL part and na namespace.

Also handles URL redirections : allows setting an Entity as “index”, which means that it will become the default routed entity for the parent entity (implementation details in `get_redirect_pattern()`).



redirect_max_depth = 100

Attribute redirect_max_depth Max depth when following redirect attributes

__init__ (`namespace=None`, `url_part=None`, `redirect=None`, `add_redirect=None`, `add_redirect_silent=None`, `get_redirect_silent=None`, `generic=None`, `**kwargs`)

Initialize Router base attributes from given arguments

Parameters

- **namespace** – Optional. See `namespace`
- **url_part** – Optional. See `url_part`
- **redirect** – Optional. See `redirect`
- **add_redirect** – Optional. See `add_redirect`
- **add_redirect_silent** – Optional. See `add_redirect_silent`
- **get_redirect_silent** – Optional. See `get_redirect_silent`
- **generic** – Optional. See `generic`

namespace = None

Attribute namespace If defined, group this router’s patterns in an URL namespace

url_part = None

Attribute url_part If defined, add to router URL (use when as regex when building URL group)

redirect = None

Attribute `redirect` If defined, `Router` will add a redirect view to the returned patterns. To get the redirect target, `get_redirect_pattern()` will follow `redirect` attributes in the stored entities. The attribute's value is altered by the `register()`, if `index` is `True` in its arguments or if the registered entity `django_crucrudile.entities.Entity.index` attribute is set to `True`.

`add_redirect = None`

Attribute `add_redirect` Add redirect pattern when calling `patterns()`. If `None` (default), will be guessed using `redirect` (Add redirect only if there is one defined)

`add_redirect_silent = False`

Attribute `add_redirect_silent` Fail silently when the patterns reader is asked to add the redirect patterns and the redirect attribute is not set (on self). Defaults to `False`, because in the default configuration, `add_redirect` is guessed using `redirect`, using `bool`. Set to `True` if you're using `add_redirect` explicitly and want the redirect pattern to be optional.

`get_redirect_silent = False`

Attribute `get_redirect_silent` Fail silently when following redirect attributes to find the redirect URL name (if no URL name is found).

`generic = False`

Attribute `generic` If `True`, `get_register_map()` will return a `model.generic.GenericModelRouter` (with preconfigured Django views) for the `Model` type.

`get_register_map()`

Add two base register mappings (to the mappings returned by the super implementation)

- `django.db.models.Model` subclasses are passed to a `model.ModelRouter` (or `model.generic.GenericModelRouter`) if `generic` is set to `True`
- `django.views.generic.View` subclasses are passed to a `View`

returns Register mappings

rtype dict

`register(entity, index=False, map_kwargs=None)`

Register routed entity, using `django_crucrudile.entities.store.EntityStore.register()`

Set as index when `index` or `entity.index` is `True`.

Parameters

- **`entity`** (`django_crucrudile.entities.Entity`) – Entity to register
- **`index`** (`bool`) – Register as index (set `redirect` to `entity`)
- **`map_kwargs`** (`dict`) – Optional. Keyword arguments to pass to mapping value if entity gets transformed.

```
>>> from mock import Mock
>>> router = Router()

>>> entity = Mock()
>>> entity.index = False
>>>
>>> router.register(entity)
```

```

>>> router.redirect is None
True

>>> entity = Mock()
>>> entity.index = False
>>>
>>> router.register(entity, index=True)
>>> router.redirect is entity
True

>>> entity = Mock()
>>> entity.index = True
>>>
>>> router.register(entity)
>>> router.redirect is entity
True

```

get_redirect_pattern (*namespaces=None, silent=None, redirect_max_depth=None*)

Compile the URL name to this router's redirect path (found by following `Router.redirect`), and that return a lazy `django.views.generic.RedirectView` that redirects to this URL name

Parameters

- **namespaces** (*list of str*) – Optional. The list of namespaces will be used to get the current namespaces when building the redirect URL name. If not given an empty list will be used.
- **silent** (*bool*) – Optional. See `Router.get_redirect_silent`
- **redirect_max_depth** (*int*) – Optional. See `Router.redirect_max_depth`

Raises

- **OverflowError** – If the depth-first search in the graph made from redirect attributes reaches the depth in `redirect_max_depth` (to intercept graph cycles)
- **ValueError** – If no redirect found when following redirect attributes, and silent mode is not enabled.

```

>>> from mock import Mock
>>> entity = Mock()
>>> entity.redirect.redirect = 'redirect_target'
>>>
>>> router = Router()
>>> router.register(entity)
>>>
>>> pattern = router.get_redirect_pattern()
>>>
>>> type(pattern).__name__
'RegexURLPattern'
>>> pattern.callback.__name__
'RedirectView'
>>> pattern._target_url_name
'redirect_target'

>>> from mock import Mock
>>> entity = Mock()
>>> entity.redirect.redirect = 'redirect_target'
>>>
>>> router = Router()
>>> router.register(entity)
>>>

```

```
>>> pattern = router.get_redirect_pattern(
...     namespaces=['ns1', 'ns2']
... )
>>> type(pattern).__name__
'RegexURLPattern'
>>> pattern.callback.__name__
'RedirectView'
>>> pattern._target_url_name
'ns1:ns2:redirect_target'

>>> entity = Mock()
>>> entity.redirect.redirect = entity
>>>
>>> router = Router()
>>> router.register(entity)
>>>
>>> router.get_redirect_pattern()
...
Traceback (most recent call last):
...
OverflowError: Depth-first search reached its maximum (100)
depth, without returning a leaf item (string). Maybe the
redirect graph has a cycle ?

>>> entity = Mock()
>>> entity.__str__ = lambda x: 'mock redirect'
>>> entity.redirect = None
>>>
>>> router = Router()
>>> router.register(entity)
>>>
>>> router.get_redirect_pattern()
...
Traceback (most recent call last):
...
ValueError: Failed following redirect attribute (mock
redirect) (last redirect found : mock redirect, redirect
value: None) in Router
```

patterns (*namespaces=None, add_redirect=None, add_redirect_silent=None*)

Read `_store` and yield a pattern of an URL group (with url part and namespace) containing entities's patterns (obtained from the entity store), also yield redirect patterns where defined.

Parameters

- **namespaces** (*list of str*) – We need `patterns()` to pass namespaces recursively, because it may be needed to make redirect URL patterns
- **add_redirect** (*bool*) – Override `Router.add_redirect`
- **add_redirect_silent** – Override `Router.add_redirect_silent`

```
>>> from mock import Mock
>>> router = Router()
>>> pattern = Mock()

>>> entity_1 = Mock()
>>> entity_1.index = False
>>> entity_1.patterns = lambda *args: ['MockPattern1']
>>>
```



```
>>> router.register(entity_1)
>>>
>>> list(router.patterns())
[<RegexURLResolver <str list> (None:None) ^>]
>>> next(router.patterns()).url_patterns
['MockPattern1']

>>> entity_2 = Mock()
>>> entity_2.index = True
>>> entity_2.redirect = 'redirect_target'
>>> entity_2.patterns = lambda *args: ['MockPattern2']
>>>
>>> router.register(entity_2)
>>>
>>> list(router.patterns())
...
[<RegexURLResolver <RegexURLPattern list> (None:None) ^>]
>>> next(router.patterns()).url_patterns
...
[<RegexURLPattern redirect_target-redirect ^$>,
 'MockPattern1', 'MockPattern2']

>>> router.redirect = None
>>>
>>> list(router.patterns(add_redirect=True))
...
Traceback (most recent call last):
...
ValueError: No redirect attribute set (and
'add_redirect_silent' is 'False').
```

3.4.2 Router mixins

Model router mixin

class django_crucrudile.routers.mixins.model.**ModelMixin** (*model=None, url_part=None, **kwargs*)

Bases: builtins.object

ModelRouter with no views. Give `model` kwarg where needed, ask it in `__init__()`, and map `SingleObjectMixin` and `MultipleObjectMixin` to `django_crucrudile.routes.ModelViewRoute` in register functions.

django_crucrudile.routers.mixins.model.ModelMixin

`__init__` (*model=None, url_part=None, **kwargs*)

Read model (from args or class-level value (`model`), fail if none found.

Parameters `model` (`django.db.Models`) – see `model`

Raises ValueError if model not passed an argument and not defined on class

model = None

Attribute model Model used when building router URL name and URL part, and passed to registered routes. Must be defined at class-level or passed to `__init__()`.

model_url_part

Return the model name to be used when building the URL part

Returns URL part from the Django model name (`model._meta.model_name`)

Return type str

```
>>> from mock import Mock
>>>
>>> model = Mock()
>>> model._meta.model_name = 'testmodel'
>>> route = ModelMixin(model=model)
>>>
>>> route.model_url_part
'testmodel'
```

get_register_map_kwargs()

Add `model` as kwarg when applying register map

Returns Keyword arguments to pass to mapping value, when applying register map (from `get_register_map()` in `register()`)

Return type dict

See also:

For doctests that use this member, see `django_crucrudile.routers.model.ModelRouter`

get_base_store_kwargs()

Add `model` so that the route classes in the base store will get the model as a kwarg when being instantiated

Returns Keyword arguments to pass to mapping value, when applying class register map (from `get_register_class_map()` in `register_class()`)

Return type dict

See also:

For doctests that use this member, see `django_crucrudile.routers.model.ModelRouter`

get_register_map()

Add `django_crucrudile.routes.ModelViewRoute` mapping for `SingleObjectMixin` and `MultipleObjectMixin`.

This mapping allows registering Django generic views in the base, making them register as `django_crucrudile.routes.ModelViewRoute` and instantiate with `model`.

Returns Entity store mappings

Return type dict

See also:

For doctests that use this member, see `django_crucrudile.routers.model.ModelRouter`

classmethod get_register_class_map()

Add `django_crucrudile.routes.ModelViewRoute.make_for_view()` mapping for `SingleObjectMixin` and `MultipleObjectMixin`.

This mapping allows registering Django generic views in the base store, so that entities made using `django_crucrudile.routes.ModelViewRoute` (and instantiated with `model`) get registered in the entity store when `ModelMixin` gets instantiated (in `django_crucrudile.entities.store.EntityStore.register_base_store()`).

`django_crucrudile.routes.ModelViewRoute.make_for_view()` creates a new `django_crucrudile.routes.ModelViewRoute` class, and uses its argument as the `django_crucrudile.routes.ViewRoute.view_class` attribute.

Returns Base store mappings

Return type dict

See also:

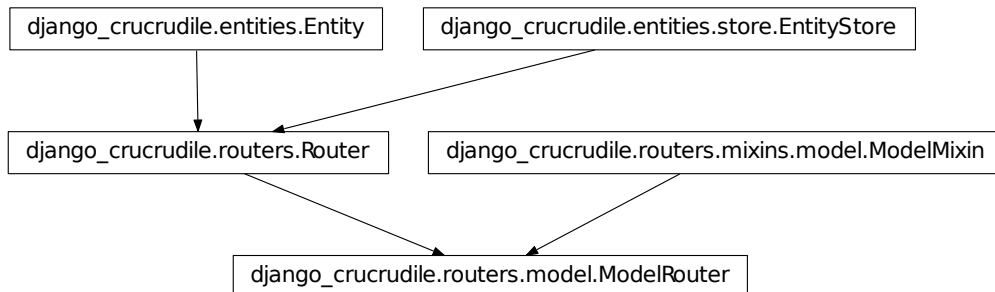
For doctests that use this member, see `django_crucrudile.routers.model.ModelRouter`

3.4.3 Model router

class `django_crucrudile.routers.model.ModelRouter` (*model=None*, *url_part=None*, ***kwargs*)

Bases: `django_crucrudile.routers.mixins.model.ModelMixin`, `django_crucrudile.routers.Router`

Model router, implements `django_crucrudile.routers.mixins.model.ModelMixin` with :class: `django_crucrudile.routers.Router`, to provide a Model that passes the model when instantiating entities.



```

>>> from django.views.generic.detail import SingleObjectMixin
>>> from mock import Mock
>>>
>>> class GenericView(SingleObjectMixin):
...     pass
>>> class NotGenericView:
...     pass

>>> model = Mock()
>>> view = Mock()
>>>
>>> model._meta.model_name = 'modelname'
>>>
>>> router = ModelRouter(model=model)
  
```

```
>>>
>>> router.model_url_part
'modelname'

>>> router.register(GenericView)
```

Generic model router

class `django_crucrudile.routers.model.generic.GenericModelRouter` (*model=None*,
url_part=None,
***kwargs*)

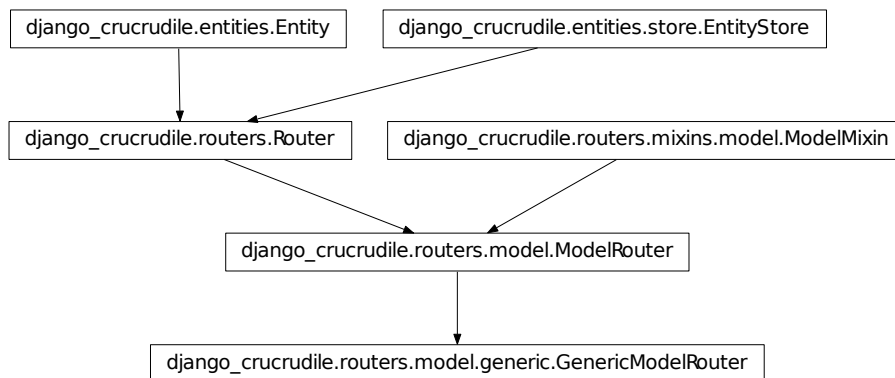
Bases: `django_crucrudile.routers.model.ModelRouter`

Generic model router, subclasses `django_crucrudile.routers.model.ModelRouter` and use 5 Django generic views for each registered model.

Provides specific `django_crucrudile.routes.ModelViewRoute` classes, created for the following Django generic views :

- `django.views.generic.ListView`
- `django.views.generic.DetailView`
- `django.views.generic.CreateView`
- `django.views.generic.UpdateView`
- `django.views.generic.DeleteView`

These classes are registered in the base store, using `django_crucrudile.entities.store.EntityStore.register` or the `django_crucrudile.entities.store.provides()` decorator. They will be instantiated (with the model as argument) when the router is itself instantiated, using `django_crucrudile.entities.store.EntityStore.register_base_store()`.



```
>>> import tests.unit
>>> from django.db.models import Model
>>> from django_crucrudile.routers import Router, ModelRouter
>>>
>>> # needed to subclass Django Model
>>> __name__ = "tests.doctests"
>>>
```

```

>>> class TestModel(Model):
...     pass

>>> router = Router(generic=True)
>>>
>>> router.register(TestModel)

>>> print(router.get_str_tree())
...
- Router @ ^
- GenericModelRouter testmodel @ ^testmodel/
- testmodel-list-redirect @ ^$ RedirectView
- testmodel-delete @ ^delete$ DeleteView
- testmodel-update @ ^update$ UpdateView
- testmodel-create @ ^create$ CreateView
- testmodel-detail @ ^detail$ DetailView
- testmodel-list @ ^list$ ListView

```

The following graph may help to explain the relation between the generic views, routes and routers :

3.5 URL utils

Contents

- URL utils
 - Decorators
 - Functions
 - Classes
 - * Generic
 - * URL parts classes
 - Optional URL parts list
 - URL Builder

This module contains some utility modules for handling URL building, and the aspect of handling several parts of the URL, each separated by different separators, that may be provided or not (thus, handling separators becomes a bit more complicated).

`django_crucrudile.urlutils.pass_tuple(count=1)`

Returns a decorator that wraps a function to make it run without the first part of a tuple in its original arguments and return the omitted arguments concatenated with its original return value.

Parameters `count` (*int*) – Number of arguments to omit, default 1.

Returns Decorator

Return type function

Warning: This function is not the actual decorator, but a function that returns that decorator (with the given tuple slice index). If it is used as a decorator, it should be written `@pass_tuple()` instead of `@pass_tuple`.

```

>>> pack_with_42 = lambda x: (42, x)
>>> pack_with_42(8)
(42, 8)

```

```
>>> add_2 = pass_tuple()(lambda x: x + 2)
>>> add_2(pack_with_42(8))
(42, 10)
>>> mul_2 = pass_tuple()(lambda x: x*2)
>>> mul_2(add_2(pack_with_42(8)))
(42, 20)
>>> unpack = lambda x: x[0] + x[1]
>>> unpack(mul_2(add_2(pack_with_42(8))))
62
```

`django_crucrudile.urlutils.compose` (*functions*, *args, **kwargs)
Compose functions together

Parameters *functions* (list of callables) – Functions to compose

Returns Composed function

Return type function

Note: This function will pass all other arguments and keyword arguments to the composed functions.

```
>>> compose([lambda x: x*2, lambda x: x+2])(5)
12
```

class `django_crucrudile.urlutils.Separated` (*args, separator=None, opt_separator=None, required_default=None, **kwargs)
Bases: `builtins.object`

Accepts separator options in `__init__()`, and provide `get_separator()`, that returns the corresponding separator for required and optional parts, based on the separator passed to `__init__()` (or set at class-level).

`django_crucrudile.urlutils.Separated`

separator = '/'

Attribute separator Separator to use in front of a required item

opt_separator = '/?'

Attribute opt_separator Separator to use in front of an optional item

required_default = True

Attribute required_default If True, items required by default (when None)

get_separator (*required=None*)

Get the argument separator to use according to the *required* argument

Parameters *required* (*bool*) – If False, will return the optional argument separator instead of the regular one. Default is True.

Returns Separator

Return type str

```
>>> Separated().get_separator()
'/'
>>> Separated().get_separator(True)
'/'
>>> Separated().get_separator(False)
'/?'
```

class django_crucrudile.urlutils.**Parsable**

Bases: builtins.object

Class whose instances may be called, to return a “parsed” version, obtained by passing the original version in the parsers returned by `get_parsers()`.

django_crucrudile.urlutils.Parsable

```
>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda x: x*2, lambda x: x+2]
>>>
>>> TestParsable(5) ()
12

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return []
>>>
>>> TestParsable(5) ()
5

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda x: None]
>>>
>>> TestParsable(5) ()

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [None]
>>>
>>> TestParsable(5) ()
Traceback (most recent call last):
...
TypeError: the first argument must be callable

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda x, y: None]
>>>
>>> TestParsable(5) ()
Traceback (most recent call last):
```

```

...
TypeError: <lambda>() missing 1 required positional argument: 'y'

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda: None]
>>>
>>> TestParsable(5)()
Traceback (most recent call last):
...
TypeError: <lambda>() takes 0 positional arguments but 1 was given

```

get_parsers()

Return parsers list. Base implementation returns an empty list. To add new parsers, override this function and append/prepend the functions to use as parsers.

Returns List of parser functions

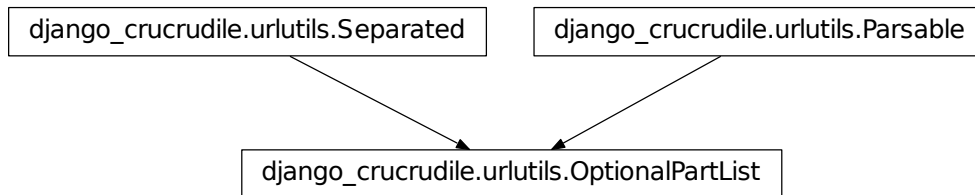
Return type list

class django_crucrudile.urlutils.**OptionalPartList** (*iterable=None, separator=None, opt_separator=None, required_default=None*)

Bases: django_crucrudile.urlutils.Separated, django_crucrudile.urlutils.Parsable, builtins.list

Implement Separated and Parsable into a list, to make a separated, parsable URL part list, that handles optional parts and that uses registered parsers (from `get_parsers()`) when the instance is called.

Provide two base parsers, that convert, if needed, original items in 2-tuples (`transform_to_tuple()`), and provide a default value for the first item of the tuple if it's None (`apply_required_default()`).



```

>>> builder = OptionalPartList(
...     ["<1>", (None, "<2>"), (False, "<3>")]
... )
>>>
>>> list(builder())
[(True, '<1>'), (True, '<2>'), (False, '<3>')]

>>> failing_builder = OptionalPartList(
...     ["<1>", (None, "<2>"), (False, "<3>", "fail")]
... )
>>>
>>> list(failing_builder())
Traceback (most recent call last):
...

```


`ValueError`: too many values to unpack (expected 2)

get_parsers()

Complement `OptionalPartList` parsers (from `OptionalPartList.get_parsers()`) with `transform_to_tuple()` and `apply_required_default()`.

Returns List of parser functions

Return type list

static transform_to_tuple(items)

Transform each item to a tuple if it's not one

Parameters *items* (*iterable*) – List of items and tuples

Returns List of tuples

Return type iterable of tuple

```
>>> list(OptionalPartList.transform_to_tuple([
...     '<1>',
...     (None, '<2>')
... ]))
[(None, '<1>'), (None, '<2>')]
```

static apply_required_default(items, default)

Apply default value to first element of item if it's None.

Parameters

- **items** (*iterable*) – List of tuples
- **default** (*boolean*) – Value to use if none provided

Returns List of tuples, with required default value applied

Return type iterable of tuple

```
>>> list(
...     OptionalPartList.apply_required_default(
...         [
...             ('<provided>', '<1>'),
...             (None, '<2>')
...         ],
...         default='<default>'
...     )
... )
[('<provided>', '<1>'), ('<default>', '<2>')]
```

class `django_crucrudile.urlutils.URLBuilder` (*iterable=None*, *separator=None*,
opt_separator=None, *required_default=None*)

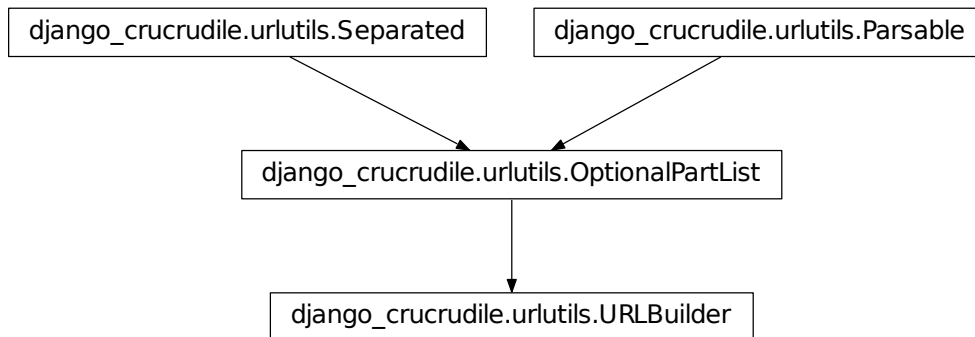
Bases: `django_crucrudile.urlutils.OptionalPartList`

Allows building URLs from a list of URL parts. The parts can be required or optional, this information will be used to determine which separator to use.

We subclass `OptionalPartList`, and add our parsers in `get_parsers()`, so that they are used when the instance gets called :

- `filter_empty_items()`
- `add_first_item_required_flag()`
- `flatten()`

•join()



```

>>> builder = URLBuilder(
...     ["<1>", (False, "<2>"), (True, "<3>")]
... )
>>>
>>> builder()
(True, '<1>/?<2>/<3>')

>>> builder = URLBuilder(
...     ["<1>", "<2>", (False, "<3>")]
... )
>>>
>>> builder()
(True, '<1>/<2>/?<3>')

>>> builder = URLBuilder(
...     [(False, "<1>"), "<2>", (False, "<3>")]
... )
>>>
>>> builder()
(False, '<1>/<2>/?<3>')

>>> builder = URLBuilder(
...     [(False, "<1>"), None, (True, None)]
... )
>>>
>>> builder()
(False, '<1>')

>>> builder = URLBuilder(
...     [(False, "<1>"), 1]
... )
>>>
>>> builder()
Traceback (most recent call last):
...
TypeError: sequence item 2: expected str instance, int found

get_parsers()
  
```

Complement `OptionalPartList` parsers (from `OptionalPartList.get_parsers()`) with `filter_empty_items()`, `add_first_item_required_flag()`, `flatten()` and `join()`.

Returns List of parser functions

Return type list

static filter_empty_items (*items*)

Filter out items that give False when casted to boolean.

Parameters *items* (*iterable*) – List of tuples

Returns List of URL part specs (with empty items cleared out)

Return type list of tuple

```
>>> list(URLBuilder.filter_empty_items([
...     (None, ''),
...     (None, '<not empty>'),
...     (None, []),
...     (None, None),
...     (None, '<not empty 2>'),
... ]))
[(None, '<not empty>'), (None, '<not empty 2>')]

>>> list(URLBuilder.filter_empty_items([
...     (None, '<not empty>'),
...     None
... ]))
Traceback (most recent call last):
...
TypeError: 'NoneType' object is not iterable
```

static add_first_item_required_flag (*items*)

Return a boolean indicating whether the first item is required, and the list of items.

Parameters *items* (*iterable*) – List of tuples

Returns Tuple with “first item required” flag, and item list

Return type tuple : (boolean, list)

```
>>> output = URLBuilder.add_first_item_required_flag(
...     [(False, '<opt>'), (True, '<req>')]
... )
>>>
>>> output[0], list(output[1])
(False, [(False, '<opt>'), (True, '<req>')])

>>> output = URLBuilder.add_first_item_required_flag(
...     []
... )
>>>
>>> output[0], list(output[1])
(False, [])

>>> output = URLBuilder.add_first_item_required_flag(
...     [(None, ) * 3]
... )
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 2)
```

static `flatten(items, get_separator)`

Flatten items, adding the separator where required.

Parameters `items` (*iterable*) – List of tuples

Returns List of URL parts with separators

Return type iterable of str

Warning: This function is decorated using `pass_tuple()`, the first part of the tuple in its arguments will be omitted, and inserted at the beginning of the return value, automatically. See the documentation of `pass_tuple()` for more information.

```
>>> get_separator = lambda x: '/'

>>> output = URLBuilder.flatten(
...     (None, [(True, '<1>'), (True, '<2>')]),
...     get_separator
... )
>>>
>>> output[0], list(output[1])
(None, ['<1>', '/', '<2>'])

>>> from mock import Mock
>>>
>>> get_separator = Mock()
>>> get_separator.side_effect = ['/']

>>> output = URLBuilder.flatten(
...     (None, [(True, '<1>'), (True, '<2>')]),
...     get_separator
... )
>>>
>>> output[0], list(output[1])
(None, ['<1>', '/', '<2>'])
>>> get_separator.assert_called_once_with(True)
```

static `join(items)`

Concatenate items into a string

Parameters `items` (*list of str*) – List of URL parts, with separators

Returns Joined URL parts

Return type str

Warning: This function is decorated using `pass_tuple()`, the first part of the tuple in its arguments will be passed automatically. See the documentation of `pass_tuple()` for more information.

```
>>> URLBuilder.join((None, ['a', 'b']))
(None, 'ab')

>>> URLBuilder.join((None, [['a'], 'b']))
Traceback (most recent call last):
...
TypeError: sequence item 0: expected str instance, list found
```

```
>>> URLBuilder.join((None, ['a', None]))
Traceback (most recent call last):
...
TypeError: sequence item 1: expected str instance, NoneType found
```

3.5.1 Decorators

This module defines the `pass_tuple()` decorator, that makes a function :

- run without the first part of its original arguments
- return the omitted arguments and its original return value

This allows to use chain of functions in `Parsable` that work only on a slice of the arguments (for example, to run some operations while passing a boolean flag).

`django_crucrudile.urlutils.pass_tuple(count=1)`

Returns a decorator that wraps a function to make it run without the first part of a tuple in its original arguments and return the omitted arguments concatenated with its original return value.

Parameters `count` (*int*) – Number of arguments to omit, default 1.

Returns Decorator

Return type function

Warning: This function is not the actual decorator, but a function that returns that decorator (with the given tuple slice index). If it is used as a decorator, it should be written `@pass_tuple()` instead of `@pass_tuple`.

```
>>> pack_with_42 = lambda x: (42, x)
>>> pack_with_42(8)
(42, 8)
>>> add_2 = pass_tuple()(lambda x: x + 2)
>>> add_2(pack_with_42(8))
(42, 10)
>>> mul_2 = pass_tuple()(lambda x: x*2)
>>> mul_2(add_2(pack_with_42(8)))
(42, 20)
>>> unpack = lambda x: x[0] + x[1]
>>> unpack(mul_2(add_2(pack_with_42(8))))
62
```

3.5.2 Functions

This module defines the `compose()` function, that compose a list of functions into a single function that returns its arguments, passed in chain to each of the functions. This function is used by `Parsable` to compose the parsers returned by `Parsable.get_parsers()`, in `Parsable.__call__()`.

`django_crucrudile.urlutils.compose(functions, *args, **kwargs)`

Compose functions together

Parameters `functions` (*list of callables*) – Functions to compose

Returns Composed function

Return type function

Note: This function will pass all other arguments and keyword arguments to the composed functions.

```
>>> compose([lambda x: x*2, lambda x: x+2])(5)
12
```

3.5.3 Classes

- `Separated` allows separator-related options to be passed to `Separated.__init__()` and provides `Separated.get_separator()`
- `Parsable` provides a class which instances can be called (`Parsable.__call__()`), to return the “parsed” version of the instance. The parsed version is made by passing the instance through the functions returned by `Parsable.get_parsers()`
- `OptionalPartList` provides a class that implements `Separated` and `Parsable` with a list, and that provides two parsers (that, if needed : transform the original items in tuples ; set the “required” flag to the default value)
- `URLBuilder` subclasses `OptionalPartList` and provides parsers , on top of the original ones, to join the URL parts with adequate separators where required

Generic

Note: These classes provide bases for the `URLBuilder` and `django_crucrudile.routes.mixins.arguments.parser.` classes.

```
class django_crucrudile.urlutils.Separated(*args, separator=None, opt_separator=None, re-
                                         quired_default=None, **kwargs)
```

Bases: `builtins.object`

Accepts separator options in `__init__()`, and provide `get_separator()`, that returns the corresponding separator for required and optional parts, based on the separator passed to `__init__()` (or set at class-level).

django_crucrudile.urlutils.Separated

```
__init__(*args, separator=None, opt_separator=None, required_default=None, **kwargs)
```

Initialize, set separator options

Parameters

- **separator** – See `separator`
- **opt_separator** – See `opt_separator`
- **required_default** – See `required_default`

separator = `'/'`

Attribute separator Separator to use in front of a required item

`opt_separator = '/'`

Attribute `opt_separator` Separator to use in front of an optional item

`required_default = True`

Attribute `required_default` If True, items required by default (when None)

`get_separator(required=None)`

Get the argument separator to use according to the `required` argument

Parameters `required` (*bool*) – If False, will return the optional argument separator instead of the regular one. Default is True.

Returns Separator

Return type str

```
>>> Separated().get_separator()
'/'
>>> Separated().get_separator(True)
'/'
>>> Separated().get_separator(False)
'/?'
```

class `django_crucrudile.urlutils.Parsable`

Bases: `builtins.object`

Class whose instances may be called, to return a “parsed” version, obtained by passing the original version in the parsers returned by `get_parsers()`.

django_crucrudile.urlutils.Parsable

```
>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda x: x*2, lambda x: x+2]
>>>
>>> TestParsable(5) ()
12

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return []
>>>
>>> TestParsable(5) ()
5

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda x: None]
>>>
>>> TestParsable(5) ()
```

```
>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [None]
>>>
>>> TestParsable(5)()
Traceback (most recent call last):
...
TypeError: the first argument must be callable

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda x, y: None]
>>>
>>> TestParsable(5)()
Traceback (most recent call last):
...
TypeError: <lambda>() missing 1 required positional argument: 'y'

>>> class TestParsable(Parsable, int):
...     def get_parsers(self):
...         return [lambda: None]
>>>
>>> TestParsable(5)()
Traceback (most recent call last):
...
TypeError: <lambda>() takes 0 positional arguments but 1 was given
```

get_parsers()

Return parsers list. Base implementation returns an empty list. To add new parsers, override this function and append/prepend the functions to use as parsers.

Returns List of parser functions

Return type list

__call__()

Compose the parsers in `get_parsers()` using `compose()`, and use the composed function to get the parsed version from the original version.

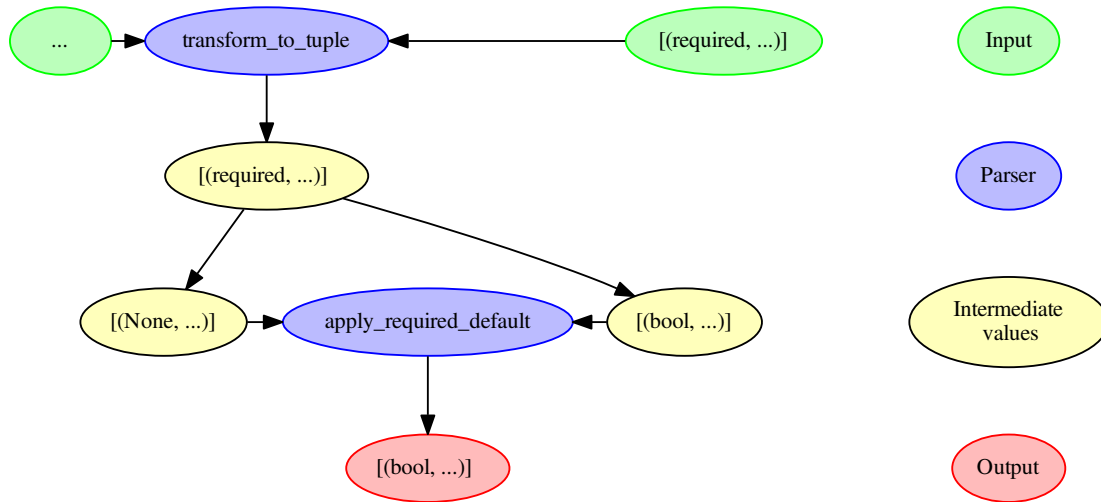
Returns output of parsers

See also:

For doctests that use this member, see `Parsable`

URL parts classes

Optional URL parts list

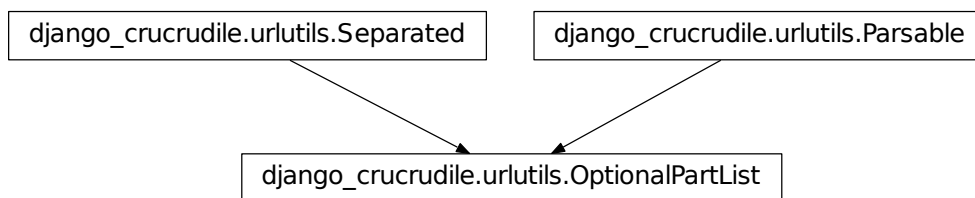


class `django_crucrudile.urlutils.OptionalPartList` (*iterable=None, separator=None, opt_separator=None, required_default=None*)

Bases: `django_crucrudile.urlutils.Separated`, `django_crucrudile.urlutils.Parsable`, `builtins.list`

Implement `Separated` and `Parsable` into a list, to make a separated, parsable URL part list, that handles optional parts and that uses registered parsers (from `get_parsers()`) when the instance is called.

Provide two base parsers, that convert, if needed, original items in 2-tuples (`transform_to_tuple()`), and provide a default value for the first item of the tuple if it's `None` (`apply_required_default()`).



```

>>> builder = OptionalPartList(
...     ["<1>", (None, "<2>"), (False, "<3>")]
... )
>>> list(builder())
[(True, '<1>'), (True, '<2>'), (False, '<3>')]

```

```
>>> failing_builder = OptionalPartList(
...     ["<1>", (None, "<2>"), (False, "<3>", "fail")]
... )
>>>
>>> list(failing_builder())
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 2)
```

__add__ (*other*)

Concatenate with other iterable, creating a new object..

We override `list.__add__()` to return a new `OptionalPartList` instance, instead of a list instance.

Parameters *other* (*iterable*) – Iterable to concatenate with

Returns Concatenated object

Return type `type(self)`

```
>>> a = OptionalPartList(['foo'])
>>> b = OptionalPartList(['bar'])
>>>
>>> a + b
['foo', 'bar']
>>>
>>> type(a + b)
<class 'django_crucrudile.urlutils.OptionalPartList'>
>>>
>>> (a + b) is a
False
>>>
>>> (a + b) is b
False

>>> (a + None) is a
True
```

__init__ (*iterable=None, separator=None, opt_separator=None, required_default=None*)

Initialize, use empty list as iterable if None provided.

Parameters

- **iterable** (*iterable*) – Raw URL part list
- **separator** (*str*) – See `Separated.__init__()`
- **opt_separator** (*str*) – See `Separated.__init__()`
- **required_default** (*bool*) – See `Separated.__init__()`

get_parsers ()

Complement `OptionalPartList` parsers (from `OptionalPartList.get_parsers()`) with `transform_to_tuple()` and `apply_required_default()`.

Returns List of parser functions

Return type list

static transform_to_tuple (*items*)

Transform each item to a tuple if it's not one

Parameters **items** (*iterable*) – List of items and tuples

Returns List of tuples

Return type iterable of tuple

```
>>> list(OptionalPartList.transform_to_tuple([
...     '<1>',
...     (None, '<2>')
... ]))
[(None, '<1>'), (None, '<2>')]
```

static apply_required_default (*items, default*)

Apply default value to first element of item if it's None.

Parameters

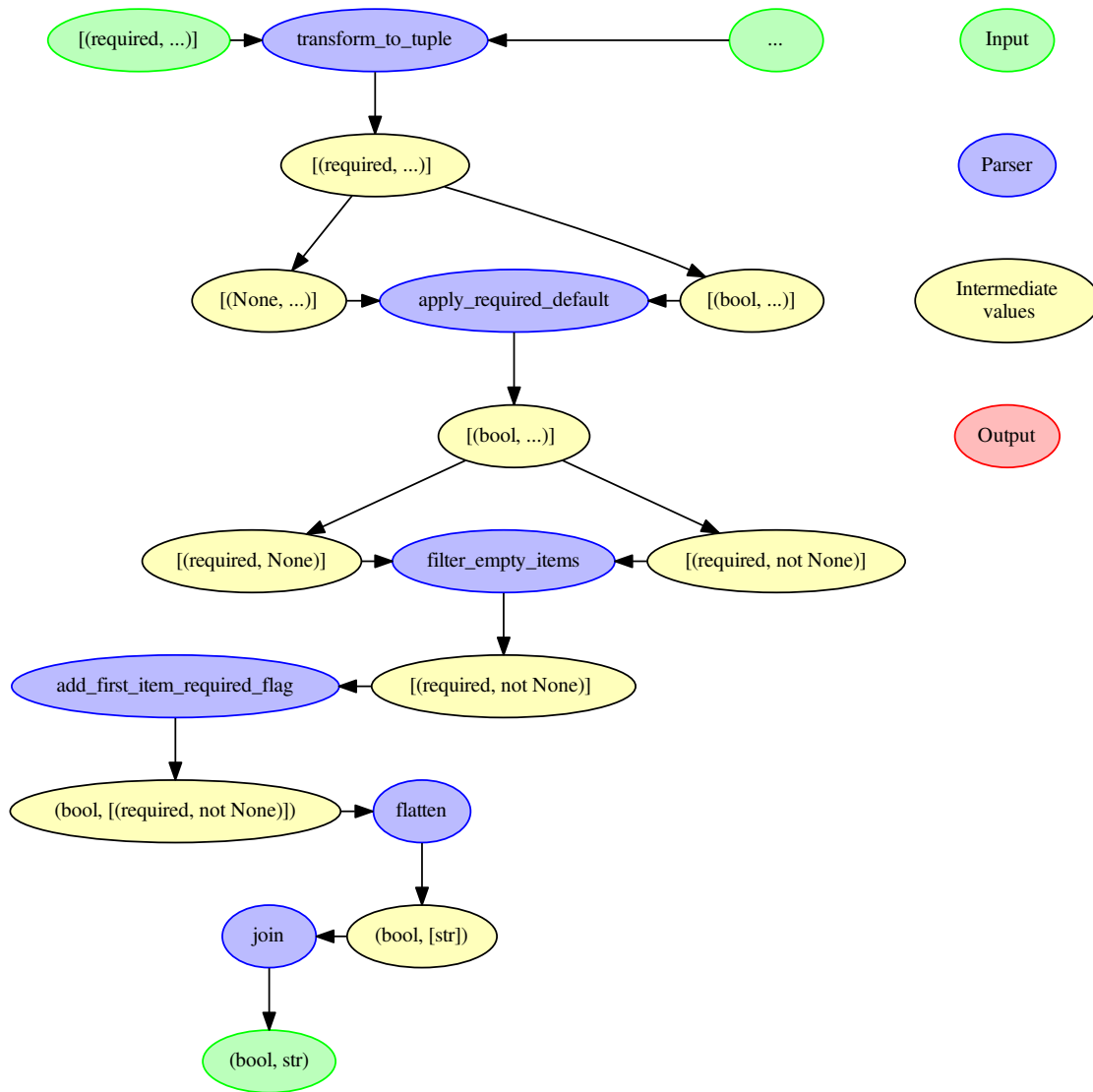
- **items** (*iterable*) – List of tuples
- **default** (*boolean*) – Value to use if none provided

Returns List of tuples, with required default value applied

Return type iterable of tuple

```
>>> list(
...     OptionalPartList.apply_required_default(
...         [
...             ('<provided>', '<1>'),
...             (None, '<2>')
...         ],
...         default='<default>'
...     )
... )
[('<provided>', '<1>'), ('<default>', '<2>')]
```

URL Builder



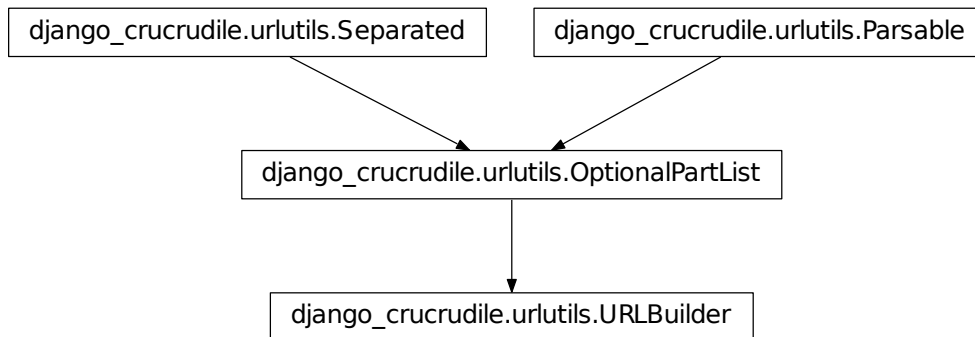
class `django_crucrudile.urlutils.URLBuilder` (*iterable=None, separator=None, opt_separator=None, required_default=None*)
 Bases: `django_crucrudile.urlutils.OptionalPartList`

Allows building URLs from a list of URL parts. The parts can be required or optional, this information will be used to determine which separator to use.

We subclass `OptionalPartList`, and add our parsers in `get_parsers()`, so that they are used when the instance gets called :

- `filter_empty_items()`
- `add_first_item_required_flag()`
- `flatten()`

•join()



```

>>> builder = URLBuilder(
...     ["<1>", (False, "<2>"), (True, "<3>")]
... )
>>>
>>> builder()
(True, '<1>/?<2>/<3>')

>>> builder = URLBuilder(
...     ["<1>", "<2>", (False, "<3>")]
... )
>>>
>>> builder()
(True, '<1>/<2>/?<3>')

>>> builder = URLBuilder(
...     [(False, "<1>"), "<2>", (False, "<3>")]
... )
>>>
>>> builder()
(False, '<1>/<2>/?<3>')

>>> builder = URLBuilder(
...     [(False, "<1>"), None, (True, None)]
... )
>>>
>>> builder()
(False, '<1>')

>>> builder = URLBuilder(
...     [(False, "<1>"), 1]
... )
>>>
>>> builder()
Traceback (most recent call last):
...
TypeError: sequence item 2: expected str instance, int found

get_parsers()
  
```

Complement `OptionalPartList` parsers (from `OptionalPartList.get_parsers()`) with `filter_empty_items()`, `add_first_item_required_flag()`, `flatten()` and `join()`.

Returns List of parser functions

Return type list

static `filter_empty_items` (*items*)

Filter out items that give False when casted to boolean.

Parameters *items* (*iterable*) – List of tuples

Returns List of URL part specs (with empty items cleared out)

Return type list of tuple

```
>>> list(URLBuilder.filter_empty_items([
...     (None, ''),
...     (None, '<not empty>'),
...     (None, []),
...     (None, None),
...     (None, '<not empty 2>'),
... ]))
[(None, '<not empty>'), (None, '<not empty 2>')]

>>> list(URLBuilder.filter_empty_items([
...     (None, '<not empty>'),
...     None
... ]))
Traceback (most recent call last):
...
TypeError: 'NoneType' object is not iterable
```

static `add_first_item_required_flag` (*items*)

Return a boolean indicating whether the first item is required, and the list of items.

Parameters *items* (*iterable*) – List of tuples

Returns Tuple with “first item required” flag, and item list

Return type tuple : (boolean, list)

```
>>> output = URLBuilder.add_first_item_required_flag(
...     [(False, '<opt>'), (True, '<req>')]
... )
>>>
>>> output[0], list(output[1])
(False, [(False, '<opt>'), (True, '<req>')])

>>> output = URLBuilder.add_first_item_required_flag(
...     []
... )
>>>
>>> output[0], list(output[1])
(False, [])

>>> output = URLBuilder.add_first_item_required_flag(
...     [(None, ) * 3]
... )
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 2)
```

static `flatten(items, get_separator)`

Flatten items, adding the separator where required.

Parameters `items` (*iterable*) – List of tuples

Returns List of URL parts with separators

Return type iterable of str

Warning: This function is decorated using `pass_tuple()`, the first part of the tuple in its arguments will be omitted, and inserted at the beginning of the return value, automatically. See the documentation of `pass_tuple()` for more information.

```
>>> get_separator = lambda x: '/'

>>> output = URLBuilder.flatten(
...     (None, [(True, '<1>'), (True, '<2>')]),
...     get_separator
... )
>>>
>>> output[0], list(output[1])
(None, ['<1>', '/', '<2>'])

>>> from mock import Mock
>>>
>>> get_separator = Mock()
>>> get_separator.side_effect = ['/']

>>> output = URLBuilder.flatten(
...     (None, [(True, '<1>'), (True, '<2>')]),
...     get_separator
... )
>>>
>>> output[0], list(output[1])
(None, ['<1>', '/', '<2>'])
>>> get_separator.assert_called_once_with(True)
```

static `join(items)`

Concatenate items into a string

Parameters `items` (*list of str*) – List of URL parts, with separators

Returns Joined URL parts

Return type str

Warning: This function is decorated using `pass_tuple()`, the first part of the tuple in its arguments will be passed automatically. See the documentation of `pass_tuple()` for more information.

```
>>> URLBuilder.join((None, ['a', 'b']))
(None, 'ab')

>>> URLBuilder.join((None, [['a'], 'b']))
Traceback (most recent call last):
...
TypeError: sequence item 0: expected str instance, list found
```

```
>>> URLBuilder.join((None, ['a', None]))
Traceback (most recent call last):
...
TypeError: sequence item 1: expected str instance, NoneType found
```

Running the tests

Contents

- Running the tests
 - Dependencies
 - Running tests using `setuptools`
 - Running tests manually
 - * Nose tests
 - * Documentation tests
 - Code coverage
 - Call graphs

This package uses unit tests and doc tests.

Warning: Documentation tests use an indentation size of 2

Note: If you are using a virtual environment, it should be activated before running the following code blocks.

- [Online tests](#) (Travis-CI)
- [Online coverage reports](#) (Coveralls)

4.1 Dependencies

Dependencies (and test dependencies) are set in the `setup.py`, and installed when the package gets installed (or when the tests are executed for test dependencies).

4.2 Running tests using `setuptools`

The test collector used in `setup.py` is the `runtests.py` script uses the nose test runner to run tests.

Warning: Documentation tests are collected as a single unit test !

- nose tests can be disabled using `NO_NOSE_TESTS=1`
- doctests can be disabled using `NO_SPHINX_TESTS=1`.

Note: The following command does not need any dependency to be installed beforehand, as they will be installed (as Python eggs, in the working directory) by `setuptools` automatically. However, you may want to install them yourself before running the tests (to avoid polluting your working directory with the Python eggs built by `setuptools`) by using `pip`, in that case use `pip install -r tests/requirements.txt`.

```
python setup.py test
```

4.3 Running tests manually

Warning: To run the tests manually, needed dependencies (`pip install -r tests/requirements.txt`) have to be available in the environment.

Warning: To run the tests manually, the `DJANGO_SETTINGS_MODULE` environment variable must be set to `tests.settings`, or to a valid settings module. (export `DJANGO_SETTINGS_MODULE=tests.settings`)

4.3.1 Nose tests

To run `nosetests` (they include the doctests), use :

```
nosetests
```

4.3.2 Documentation tests

To run doctests manually, use :

```
sphinx-build -E -c docs -b html -a docs var/docs_doctests
```

4.4 Code coverage

Coverage report is automatically executed while running nose tests. `nosetests` prints a basic coverage report, and the HTML coverage report is generated in `vars/coverage_html`.

Coverage reports can also be seen at coveralls.io/r/pstch/django-crucrudile.

4.5 Call graphs

If you install `pycallgraph` (`pip install pycallgraph`), you can use it to trace call graphs of the executed code. Sadly, it does not work with `nosetests`, but it's very easy to use it manually, for specific tests :

```
echo "  
import tests.integration.test_routers as tests  
case = tests.RouterTestCase()  
case.setUp()  
case.test_get_str_tree()  
" | pycallgraph -I django_crucrudile\* graphviz -- /dev/stdin
```

The call graph will be written to `pycallgraph.png`.

Warning: `pycallgraph` may need `GraphViz` and `pydot` to be installed (a Python 3 compatible version. At this date, it is available in bitbucket.org/prologic/pydot.)

django-crucrudile provides URL routing classes, which allows you to define your URL routing structure using `Router` and `Route` classes, and then to automatically generate an URL pattern structure.

Documentation

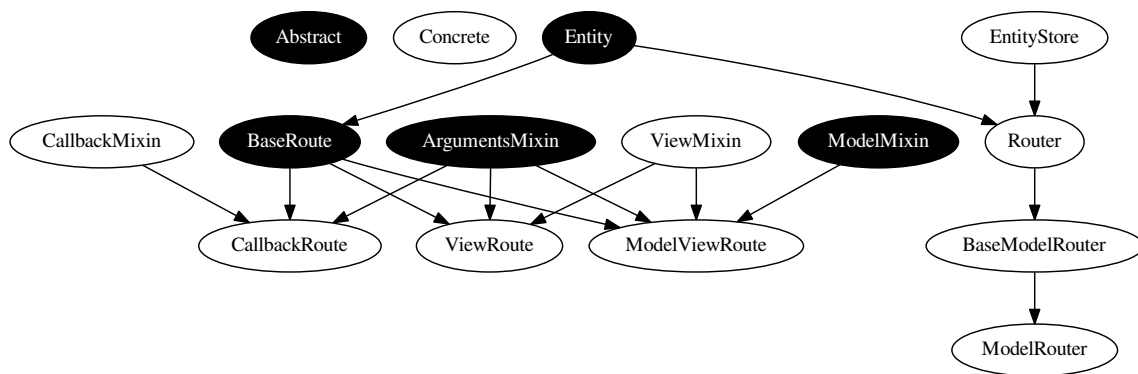
Documentation is built using Sphinx (using static reStructuredText files stored in `docs` and Sphinx-formatted docstrings in modules, classes and functions).

Use the following command to build the documentation in `docs/_build`:

```
sphinx-build -E -c docs -b html -a docs docs/_build
```

The documentation can also be viewed online, at <https://django-crucrudile.readthedocs.org/en/master/>.

Class structure graph



d

`django_crucrudile`, ??

e

`django_crucrudile.entities`, 29

`django_crucrudile.entities.store`, 22

r

`django_crucrudile.routers`, 32

`django_crucrudile.routers.mixins.model`,
37

`django_crucrudile.routers.model`, 39

`django_crucrudile.routers.model.generic`,
40

`django_crucrudile.routes`, 5

`django_crucrudile.routes.base`, 6

`django_crucrudile.routes.mixins`, 10

`django_crucrudile.routes.mixins.arguments`,
10

`django_crucrudile.routes.mixins.arguments.parser`,
12

`django_crucrudile.routes.mixins.callback`,
17

`django_crucrudile.routes.mixins.model`,
15

`django_crucrudile.routes.mixins.view`,
18

t

`tests.doctests.examples.bookstore`, 3

u

`django_crucrudile.urlutils`, 41