wheezy.routing documentation

Release latest

Andriy Kornatskyy

Apr 17, 2021

Contents

CHAPTER 1

Introduction

wheezy.routing is a python package written in pure Python code with no dependencies to other python packages. It is a simple extensible mapping between URL patterns (as plain simple strings, curly expressions or regular expressions) to a handler that can be anything you like (there is no limitation or prescription what handler is or should be).

The mapping can include other mappings and constructed dynamically.

It is optimized for performance, well tested and documented.

Resources:

- source code, examples and issues tracker are available on github
- documentation

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Install

wheezy.routing requires python version 3.6+. It is independent of operating system. You can install it from pypi site:

```
$ pip install wheezy.routing
```

2.2 Examples

Before we proceed let's setup a virtualenv environment, activate it and install:

```
$ pip install wheezy.routing
```

2.2.1 Hello World

helloworld.py shows you how to use *wheezy.routing* in a pretty simple WSGI application:

```
from wheezy.routing import PathRouter

if sys.version_info[0] >= 3:

    def ntob(n, encoding):
        return n.encode(encoding)


else:
```

(continues on next page)

(continued from previous page)

```
    def ntob(n, encoding):
        return n


def hello_world(environ, start_response):
    start_response("200 OK", [("Content-Type", "text/html")])
    yield ntob("Hello World!", "utf-8")


def not_found(environ, start_response):
    start_response("404 Not Found", [("Content-Type", "text/html")])
    yield ntob("", "utf-8")


r = PathRouter()
r.add_routes([("/", hello_world), ("/{any}", not_found)])


def main(environ, start_response):
    handler, _ = r.match(environ["PATH_INFO"])
    return handler(environ, start_response)


if __name__ == "__main__":
    from wsgiref.simple_server import make_server

    try:
        print("Visit http://localhost:8080/")
        make_server("", 8080, main).serve_forever()
    except KeyboardInterrupt:
        pass
    print("\nThanks!")
```

Let's have a look through each line in this application. First of all we import `PathRouter` that is actually just an exporting name for `PathRouter`:

```

```

Next we create a pretty simple WSGI handler to provide a response.

```
    def ntob(n, encoding):
        return n


```

In addition let's add a handler for the 'not found' response.

```
    yield ntob("Hello World!", "utf-8")


def not_found(environ, start_response):
    start_response("404 Not Found", [("Content-Type", "text/html")])
```

The declaration and mapping of patterns to handlers follows. We create an instance of `PathRouter` class and pass it a mapping, that in this particular case is a tuple of two values: `pattern` and `handler`.

```
r = PathRouter()
r.add_routes([("/", hello_world), ("/{any}", not_found)])
```

The first pattern `'/'` will match only the root path of the request (it is finishing route in the match chain). The second pattern `'/{any}'` is a curly expression, that is translated to regular expression, that ultimately matches any path and is a finishing route as well.

`main` function serves as WSGI application entry point. The only thing we do here is to get a value of WSGI environment variable `PATH_INFO` (the remainder of the request URL's path) and pass it to the router `match()` method. In return we get `handler` and `kwargs` (parameters discovered from matching rule, that we ignore for now).

```
    return handler(environ, start_response)
```

The rest in the `helloworld` application launches a simple wsgi server. Try it by running:

```
$ python helloworld.py
```

Visit http://localhost:8080/.

2.2.2 Server Time

The server time application consists of two screens. The first one has a link to the second that shows the time on the server. The second page will be mapped as a separate application with its own routing. The design used in this sample is modular. Let's start with `config` module. The only thing we need here is an instance of `PathRouter`.

```
from wheezy.routing import PathRouter

router = PathRouter()
```

The `view` module is pretty straight: a `welcome` view with a link to `server_time` view. The server time page returns the server time. And finally a catch all `not_found` handler to display http 404 error, page not found.

```
from config import router as r


def welcome(environ, start_response):
    start_response("200 OK", [("Content-type", "text/html")])
    return ["Welcome!  <a href='%s'>Server Time</a>" % r.path_for("now")]


def server_time(environ, start_response):
    start_response("200 OK", [("Content-type", "text/plain")])
    return ["The server time is: %s" % datetime.now()]


def not_found(environ, start_response):
    start_response("404 Not Found", [("Content-Type", "text/plain")])
    return ["Not Found: " + environ["routing.kwargs"]["url"]]
```

So what is interesting in the `welcome` view is a way how we get a url for `server_time` view.

```
def server_time(environ, start_response):
    start_response("200 OK", [("Content-type", "text/plain")])
```

The name `now` was used during url mapping as you can see below (module `urls`):

```
from wheezy.routing import url

server_urls = [url("time", server_time, name="now")]

all_urls = [("", welcome), ("server/", server_urls)]
all_urls += [url("{url:any}", not_found)]
```

`server_urls` are then included under the parent path `server/`, so anything that starts with the path `server/` will be directed to the `server_urls` url mapping. Lastly we add a curly expression that maps any url match to our `not_found` handler.

We combine that all together in `app` module.

```
from urls import all_urls  # noqa: I201

router.add_routes(all_urls)


def main(environ, start_response):
    handler, kwargs = router.match(environ["PATH_INFO"].lstrip("/"))
    environ["routing.kwargs"] = kwargs
    return map(
        lambda chunk: chunk.encode("utf8"), handler(environ, start_response)
    )


if __name__ == "__main__":
    from wsgiref.simple_server import make_server

    try:
        print("Visit http://localhost:8080/")
        make_server("", 8080, main).serve_forever()
    except KeyboardInterrupt:
        pass
    print("\nThanks!")
```

Try it by running:

```
$ python app.py
```

Visit http://localhost:8080/.

2.3 User Guide

2.3.1 Pattern and Handler

You create a mapping between: `pattern` (a remainder of the request URL, script name, http schema, host name or whatever else) and `handler` (callable, string, etc.):

```
urls = [
     ('posts/2003', posts_for_2003),
     ('posts/{year}', posts_by_year),
     ('posts/(?P<year>\d+)/(?P<month>\d+)', posts_by_month)
]
```

It is completely up to you how to interpret `pattern` (you can add own patterns interpretation) and/or `handler`. If you have a look at *Hello World* example you notice the following:

```
    return handler(environ, start_response)
```

or more specifically:

```
environ['PATH_INFO']
```

This operation takes the WSGI environment variable `PATH_INFO` and passes it to router for matching against available mappings. `handler` in this case is a simple callable that represents WSGI call handler.

```
    def ntob(n, encoding):
        return n
```

2.3.2 Extend Mapping

Since `mapping` is nothing more than python list, you can make any manipulation you like, e.g. add other mappings, construct them dynamically, etc. Here is snippet from *Server Time* example:

```
all_urls = [("", welcome), ("server/", server_urls)]
all_urls += [url("{url:any}", not_found)]
```

`home` mapping has been extended by simple adding another list.

2.3.3 Mapping Inclusion

Your application may be constructed with several modules, each of them can have own url mapping. You can easily include them as a `handler` (the system checks if the handler is another mapping it creates nested `PathRouter`). Here is an example from *Server Time*:

```
server_urls = [url("time", server_time, name="now")]

all_urls = [("", welcome), ("server/", server_urls)]
all_urls += [url("{url:any}", not_found)]
```

`server_urls` included into `server/` subpath. So effective path for `server_time` handler is `server/time`.

Note that the route selected for `'server/'` pattern is *intermediate* (it is not *finishing* since there is another pattern included after it). The `'time'` pattern is *finishing* since it is the last in the match chain.

2.3.4 Named Groups

Named groups are something that you can retrieve from the url mapping:

```
urls = [
    ('posts/{year}', posts_by_year),
    ('posts/(?P<year>\d+)/(?P<month>\d+)', posts_by_month)
]
```

`kwargs` is assigned a `dict` that represends key-value pairs from the match:

```
>>> handler, kwargs = r.match('posts/2011/09')
>>> kwargs
{'month': '09', 'year': '2011'}
```

2.3.5 Extra Parameters

While `named groups` get some information from the matched `path`, you can also merge these with some extra values during initialization of the mapping (this is third parameter in tuple):

```
urls = [
    ('posts', latest_posts, {'blog_id': 100})
]
```

Note, that any values from the path match override extra parameters passed during initialization.

2.3.6 `url` helper

There is *wheezy.routing.router.url()* function that let you make your url mappings more readable:

```
from wheezy.routing import url


urls = [
    url('posts', latest_posts, kwargs={'blog_id': 100})
]
```

All it does just convers arguments to a tuple of four.

2.3.7 Named Mapping

Each path mapping you create is automatically named after the handler name. The convention as to the name is: translate handler name from camel case to underscore name and remove any ending like 'handler', 'controller', etc. So `LatestPostsHandler` is named as `latest_posts`.

You can also specify an explicit name during mapping, it is convenient to use *url()*) function for this:

```
urls = [
    url('posts', latest_posts, name='posts')
]
```

When you know the name for a url mapping, you can reconstruct its path.

2.3.8 Adding Routes

You have an instance of `PathRouter`. Call its method `add_routes()` to add any pattern mapping you have. Here is how we do it in the *Hello World* example:

```
r = PathRouter()
r.add_routes([("/", hello_world), ("/{any}", not_found)])
```

... or *Server Time*:

```
from urls import all_urls  # noqa: I201

router.add_routes(all_urls)
```

2.3.9 Route Builders

Every pattern mapping you add to router is translated to an appropriate route match strategy. The available routing match strategies are definded in *config* module by `route_builders` list and include:

1. plain
2. regex
3. curly

You can easily extend this list with your own route strategies.

Plain Route

The plain route is selected in case the path satisfy the following regular expression (at least one `word`, `'/'` or `'-'` character):

```
```

The matching paths include: `account/login`, `blog/list`, etc. The strategy performs string matching.

Finishing routes are matched by exact string `equals` operation, intermediate routes are matched with `startswith` string operation.

Regex Route

Any valid regular expression will match this strategy. However there are a few limitations that apply if you would like to build paths by name (reverse function to path matching). Use regex syntax only inside named groups, create as many as necessary. The path build strategy simply replaces named groups with values supplied. Optional named groups are supported.

Curly Route

This is just a simplified version of regex routes. Curly route is something that matches the following regular expression:

```
```

You define a named group by using curly brakets. The form of curly expression (`pattern` is optional and corresponds to segment by default):

```
{name[:pattern]}
```

The curly expression `abc/{id}` is converted into regex `abc/(?P<id>[^/]+)`.

The name inside the curly expression can be constrained with the following patterns:

- `i`, `int`, `number`, `digits` - one or more digits
- `w`, `word` - one or more word characters
- `s`, `segment`, `part` - everything until `'/'` (path segment)
- `*`, `a`, `any`, `rest` - match anything

Note that if the pattern constraint doesn't correspond to anything mentioned above, then it will be interpreted as a regular expression:

```
locale:(en|ru)}/home => (?P<locale>(en|ru))/home
```

Curly routes also support optional values (these should be taken into square brackets):

```
[{locale:(en|ru)}/]home => ((?P<locale>(en|ru))/)?home
```

Here are examples of valid expressions:

```
posts/{year:i}/{month:i}
account/{name:w}
```

You can extend the recognized curly patterns:

```
from wheezy.routing.curly import patterns

patterns['w'] = r'\w+'
```

This way you can add your custom patterns.

2.3.10 Building Paths

Once you have defined routes you can build paths from them. See *Named Mapping* how the name of url mapping is constructed. Here is a example from *Server Time*:

```
def server_time(environ, start_response):
    start_response("200 OK", [("Content-type", "text/plain")])
```

You can pass optional values (`kwargs` argument) that will be used to replace named groups of the path matching pattern:

```
>>> r = RegexRoute(
...     r'abc/(?P<month>\d+)/(?P<day>\d+)',
...     kwargs=dict(month=1, day=1)
... )
>>> r.path_for(dict(month=6, day=9))
```

(continues on next page)

(continued from previous page)

```
'abc/6/9'
>>> r.path_for(dict(month=6))
'abc/6/1'
>>> r.path_for()
'abc/1/1'
```

Values passed to the `path_for()` method override any values used during initialization of url mapping.

`KeyError` is raised in case you try to build a path that doesn't exist or provide insufficient arguments for building a path.

2.4 Modules

2.4.1 wheezy.routing

`wheezy.routing.`**`url`**(*pattern*, *handler*, *kwargs=None*, *name=None*)

Converts parameters to tupple of length four. Used for convenience to name parameters and skip unused.

2.4.2 wheezy.routing.builders

`builders` module.

`wheezy.routing.builders.`**`build_route`**(*pattern*, *finishing*, *kwargs*, *name*, *route_builders*)

Try to find suitable route builder to create a route. Raises `LookupError` if none found.

2.4.3 wheezy.routing.choice

`plain` module.

`class` `wheezy.routing.choice.`**`ChoiceRoute`**(*pattern*, *finishing=True*, *kwargs=None*, *name=None*)

Route based on choice match, e.g. {locale:(en|ru)}.

`match`(*path*)

If the `path` matches, return the end of substring matched and kwargs. Otherwise return `(-1, None)`.

`path`(*values=None*)

Build the path for given route.

`wheezy.routing.choice.`**`try_build_choice_route`**(*pattern*, *finishing=True*, *kwargs=None*, *name=None*)

If the choince route regular expression match the pattern than create a ChoiceRoute instance.

2.4.4 wheezy.routing.config

`config` module.

2.4.5 wheezy.routing.curly

`curly` module.

wheezy.routing.curly.**convert**(*s*)
Convert curly expression into regex with named groups.

wheezy.routing.curly.**convert_single**(*s*)
Convert curly expression into regex with named groups.

wheezy.routing.curly.**parse**(*s*)
Parse `s` according to `group_name:pattern_name`.

There is just `group_name`, return default `pattern_name`.

wheezy.routing.curly.**replace**(*val*)
Replace `{group_name:pattern_name}` by regex with named groups.

wheezy.routing.curly.**try_build_curly_route**(*pattern*, *finishing=True*, *kwargs=None*, *name=None*)
Convert pattern expression into regex with named groups and create regex route.

2.4.6 wheezy.routing.plain

`plain` module.

class wheezy.routing.plain.**PlainRoute**(*pattern*, *finishing*, *kwargs=None*, *name=None*)
Route based on string equalty operation.

equals_match(*path*)
If the `path` exactly equals pattern string, return end index of substring matched and a copy of `self.kwargs`.

path(*values=None*)
Build the path for given route by simply returning the pattern used during initialization.

startswith_match(*path*)
If the `path` starts with pattern string, return the end of substring matched and `self.kwargs`.

wheezy.routing.plain.**try_build_plain_route**(*pattern*, *finishing=True*, *kwargs=None*, *name=None*)
If the plain route regular expression match the pattern than create a PlainRoute instance.

2.4.7 wheezy.routing.regex

class wheezy.routing.regex.**RegexRoute**(*pattern*, *finishing=True*, *kwargs=None*, *name=None*)
Route based on regular expression matching.

match_no_kwargs(*path*)
If the `path` match the regex pattern.

match_no_kwargs_finishing(*path*)
If the `path` match the regex pattern.

match_with_kwargs(*path*)
If the `path` match the regex pattern.

path_no_kwargs(*values*)
Build the path for the given route by substituting the named places of the regual expression.

Specialization case: route was initialized with no default kwargs.

path_with_kwargs(*values=None*)
Build the path for the given route by substituting the named places of the regual expression.

Specialization case: route was initialized with default kwargs.

wheezy.routing.regex.**parse_pattern**(*pattern*)

Returns path_format and names.

```
>>> parse_pattern(r'abc/(?P<id>[^/]+)')
('abc/%(id)s', ['id'])
>>> parse_pattern(r'abc/(?P<n>[^/]+)/(?P<x>\\w+)')
('abc/%(n)s/%(x)s', ['n', 'x'])
>>> parse_pattern(r'(?P<locale>(en|ru))/home')
('%(locale)s/home', ['locale'])
```

```
>>> from wheezy.routing.curly import convert
>>> parse_pattern(convert(r'[{locale:(en|ru)}/]home'))
('%(locale)s/home', ['locale'])
>>> parse_pattern(convert(r'item[/{id:i}]'))
('item/%(id)s', ['id'])
```

```
>>> p = convert('{controller:w}[/{action:w}[/{id:i}]]')
>>> parse_pattern(p)
('%(controller)s/%(action)s/%(id)s', ['controller', 'action', 'id'])
```

wheezy.routing.regex.**strip_optional**(*pattern*)

Strip optional regex group flag.

at the beginning

```
>>> strip_optional('((?P<locale>(en|ru))/)?home')
'(?P<locale>(en|ru))/home'
```

at the end

```
>>> strip_optional('item(/(?P<id>\\d+))?')
'item/(?P<id>\\d+)'
```

nested:

```
>>> p = '(?P<controller>\\w+)(/(?P<action>\\w+)(/(?P<id>\\d+))?)?'
>>> strip_optional(p)
'(?P<controller>\\w+)/(?P<action>\\w+)/(?P<id>\\d+)'
```

wheezy.routing.regex.**try_build_regex_route**(*pattern*, *finishing=True*, *kwargs=None*, *name=None*)

There is no special tests to match regex selection strategy.

2.4.8 wheezy.routing.route

route module.

class wheezy.routing.route.**Route**

Route abstract contract.

match(*path*)

if the path matches, return the end of substring matched and kwargs. Otherwise return (-1, None).

path(*values=None*)

Build the path for given route.

2.4.9 wheezy.routing.router

`router` module.

`wheezy.routing.router.`**`url`**(*pattern*, *handler*, *kwargs=None*, *name=None*)
Converts parameters to tupple of length four. Used for convenience to name parameters and skip unused.

2.4.10 wheezy.routing.utils

`utils` module.

`wheezy.routing.utils.`**`camelcase_to_underscore`**(*s*)
Convert CamelCase to camel_case.

```
>>> camelcase_to_underscore('MainPage')
'main_page'
>>> camelcase_to_underscore('Login')
'login'
```

`wheezy.routing.utils.`**`outer_split`**(*expression*, *sep='()'*)
Splits given `expression` by outer most separators.

```
>>> outer_split('123')
['123']
>>> outer_split('123(45(67)89)123(45)67')
['123', '45(67)89', '123', '45', '67']
```

If expression is not balanced raises `ValueError`.

```
>>> outer_split('123(') # doctest: +ELLIPSIS
Traceback (most recent call last):
    ...
ValueError: ...
```

`wheezy.routing.utils.`**`route_name`**(*handler*)
Return a name for the given handler. `handler` can be an object, class or callable.

```
>>> class Login: pass
>>> route_name(Login)
'login'
>>> l = Login()
>>> route_name(l)
'login'
```

`wheezy.routing.utils.`**`strip_name`**(*s*)
Strips the name per RE_STRIP_NAME regex.

```
>>> strip_name('Login')
'Login'
>>> strip_name('LoginHandler')
'Login'
>>> strip_name('LoginController')
'Login'
>>> strip_name('LoginPage')
'Login'
>>> strip_name('LoginView')
'Login'
```

(continues on next page)

(continued from previous page)

```
>>> strip_name('LoginHandler2')
'LoginHandler2'
```

Python Module Index

W

Index