simplerouter

Release 1.2

May 02, 2015

Contents

simplerouter is a simple WSGI/WebOb router partially based on the router described in WebOB's DIY Framework Tutorial.

CHAPTER 1

Contents

CHAPTER 2

Quick Example

app.py:

```
from simplerouter import Router

router = Router()
# view names are composed of modulename:function
router.add_route('/post/{name}', 'views:post_view')
router.add_route('/', 'views:index_view')

application = router.as_wsgi

if __name__=='__main__':
    from wsgiref.simple_server import make_server
    make_server('', 8000, application).serve_forever()
```

views.py:

```
from webob import Response

def post_view(request):
    post_name = request.urlvars['name']
    # ... process post_name
    return Response("Post output for %s"%post_name)

def index_view(request):
    return Response("Site index")
```

CHAPTER 3

Adding Routes

The `Router` object is composed of mappings of paths to views called routes, and are added using the `Router.add_route()` method. The route path is matched against the `Request`'s `path_info` [1] variable, and the view is either a callable, or a string indicating the location of a callable in `module_name:callable_name` format.

```
router.add_route('/path', viewfunc)
router.add_route('/path', 'module.views:named_view')
```

Route paths may contain variables, which are indicated by curly braces:

```
router.add_route('/path/{variable}/extra', viewfunc)
```

By default, path variables will match any string not containing a forward slash. Normally, a variable matches any character other than a forward slash, but an alternate regular expression can be provided after variable name with a colon character:

```
router.add_route(r'/path/{variable:\d+}', viewfunc)
```

Any path variables specified in the route path can be accessed in a dictionary attached to the `Request` object called `urlvars`:

```
def viewfunc(request):
    return Response(request.urlvars['var1'])

router.add_route('/path/{var1}/{var2}', viewfunc)
```

Path variables may also be provided via the `vars` keyword to `Router.add_route()`, which will cause them to appear in the `urlvars` dictionary. This could be useful if a view expects them but the route path doesn't contain them:

```
route.add_route('/list', viewfunc, vars={'page' : 1})
```

Routes can be added to a router on creation without needing additional `Router.add_route()` calls:

```
router = Router(
    ('/list', viewfunc, { 'vars' : {'page' : 1} }),
    ('/list/{page:\d+}', viewfunc)
)
```

[1] The path portion of a URL (the portion of the URL after the domain name) is further split into two parts called `script_name` and `path_info`. The `script_name` portion of URL indicates the path that is directly associated with the web application, and the `path_info` portion is the part of the URL after it. For a web application that is associated with an entire domain, the `script_name` would be blank, and the `path_info` would be the entire url path. It is the `path_info` that the `Router` object matches route paths against.

CHAPTER 4

Using a Router

The `Router` object is a callable that takes WebOb's `Request` object. To use it, you would construct the `Request` object from the WSGI environ, and then call the resulting `Response` object as a WSGI application:

```
def application(environ, start_response):
    # create request object
    request = Request(environ)

    # invoke router
    response = router(request)

    # complete request
    return response(environ, start_response)
```

Alternatively, the `Router.as_wsgi` method may be used to do this automatically, so long as you don't need to do any extra processing and aren't using the `Router` object within a larger framework:

```
application = router.as_wsgi
```

CHAPTER 5

Advanced Options

5.1 Default View

By default, a `Router` will return WebOb's `HTTPNotFound` error response if no view manages to return a valid response. This behavior can be changed by providing a different view via the `default` keyword to the `Router` initializer.

```
router = Router(default="module:error_view")
```

5.2 Limiting by HTTP Method

By default, view matching is not restricted by the HTTP method. The `method` keyword allows a view to be limited to specific HTTP methods, as either a single string, or a collection of strings.

Note: Views matching the GET method always also match the HEAD method.

5.3 Path Adjustment

By default, the `script_name` and `path_info` of a `Request` are not adjusted when used with a view. Normally, this wouldn't make much sense, as a route matches an entire url path, but this also makes it impossible to use a Router as a view within another Router.

To facilitate this, the `Route.add_route()` method accepts the path_info keyword, which may be a regular expression (or True, which is a synonym for the regular expression `/.*`). Matching requests are altered such that the `script_name` has the route path appended to it, and the `path_info` is replaced with the path_info keyword.

Consider the following the example:

```
example_router = Router()
example_router.add_route('/', 'example.views:index_view')
example_router.add_route('/info', 'example.views:info_view')
example_router.add_route('/help', 'example.views:help_view')

router = Router()
router.add_router('/example', example_router, path_info='/.*')
```

The following table indicates which view would be called and how the `script_name` and `path_info` would be altered:

Initial `path_info`	View	Resulting `script_name`	Resulting `path_info`
/example/	example.views:help_view	/example	/
/example/info	example.views:info_view	/example	/info
/example/help	example.views:help_view	/example	/help

5.4 Raising Responses as Exceptions

In addition to being returned normally, responses can be returned to the router by being raised by the raise statement. While this isn't usually used, this can be useful in certain circumstances, such as to prevent certain view decorators from running normally.

Only subclasses of `webob.exc.HTTPException` can be returned by being raised. Normal `Response` objects do not qualify, but all subclasses of `webob.exc.HTTPException` that have been predefined by WebOB are also `Response` objects.

5.5 Reversing paths

The `Route.reverse` method allows a path to be reversed when given the view name or the view function. If the view accepts any parameters, they can be provided to construct the URL with them.

For example:

```
router = Router()
router.add_route('/', 'example.views:index_view')
router.add_route('/help', 'example.views:help_view')
router.add_route('/get/{name}', 'example.views:get_view')

print(router.reverse('example.views:help_view'))
# "/help"

print(router.reverse('example.views:get_view', {'name' : 'duck'}))
# "/get/duck"
```

5.6 Trailing Slashes

If try_slashes is passed to the `Router` initializer, then the `Router` object will attempt to determine if a failed request would have instead succeeded if the trailing slash on the url had instead been omitted or provided. If an alternate matching route is found, then a HTTP temporary redirect response will be returned that will tell the user's browser to use the correct URL.

```
router = Router(try_slashes=True)
router.add_route('/path', viewfunc)
response = router(Request.blank('/path/'))
# response will be a redirect
```

If this option is used, it's a good idea to make sure that any views that are capable of returning `None` should opt out of this check by setting `no_alt_redir` in the `Router.add_route` registration function:

```
router.add_route('/path', viewfunc, no_alt_redir=True)
```

Under certain circumstances failure to handle this could result in an infinite redirect loop, which is why `try_slashes` is not default behavior.

5.7 View Priority

Routes are checked in the order that they are added. While this behavior is not likely to change, it still might be desirable set the priority of a route without altering the order that they are originally added, which can be done by supplying the `Router.add_route` method with the priority keyword:

```
Router.add_route('/path', viewfunc, priority=10)
```

Routes with higher number priorty values are matched against before routes with lower number priority values.

5.8 WSGI Views

A WSGI application can be provided as a view if the `wsgi` keyword is provided to the `Router.add_route` method:

```
def app_view(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return [b'hello, world\n']

router.add_route('/hello', app_view, wsgi=True)
```

Note: Most WSGI Applications do their own URL processing, so the `wsgi` keyword implies the `path_info` keyword as described in Path Adjustment. The implicitly enabled `path_info` handling can be turned off by passing `path_info=False` to `Router.add_route()`.

CHAPTER 6

Further Reading

- PEP3333 (WSGI Specification)
- WebOb documentation