antfarm Documentation

Release 0.0.1

Curtis Maloney

June 13, 2014

Contents

CHAPTER 1

Overview

Antfarm is an ultra-light weight WSGI web framework.

Essentially, it wraps the WSGI call structure, providing helpful wrappers for common needs.

You create an App instance, with a root view. A “view” is a function which accepts a `Request` instance, and returns a `Response`.

CHAPTER 2

QuickStart

Into test.py place:

```
from antfarm import App, Response

application = App(root_view = lambda r: Response('Hello World!'))
```

And launch:

```
gunicorn test:application
```

CHAPTER 3

Contents:

3.1 The App class

The root of Antfarm is the `antfarm.Ant` class.

class **`App`** (*root_view*, ***kwargs*)

> **`root_view`**
> This provides the view to call to handle all requests.
>
> Any extra kwargs will be stored as self.opts

Each Antfarm application is an App instance. Its configuration is passed to the constructor, and the instance is a callable complying with the WSGI interface (PEP3333).

3.2 Request

The `Request` class encapsulates a request, as well as providing commonly needed parsing, such as cookies, querystrings, and body.

class **`Request`** (*app*, *environ*)

> **`path`**
> The requested URI
>
> **`method`**
> The HTTP Verb used in this request (e.g. GET, POST, OPTIONS, etc)
>
> **`content_type`**
> The supplied content type of this request.
>
> **`content_params`**
> A dict containing any additional parameters passed in the content type header.
>
> **`raw_cookies`**
> A SimpleCookie object.
>
> **`cookies`**
> A simpler interface to raw_cookies, which is a dict of simply keys and values.
>
> **`body`**
> The raw contents of the request body.

query_data
A dict of data parsed from the query string.

request_data
If the request content is a HTTP Form, returns the parsed data.

The following attributes are lazy, and only parsed when accessed:

- raw_cookies
- cookies (reads raw_cookies)
- query_data
- body
- request_data (reads body)

3.3 Response

The response module includes the Response class, and a number of utilities.

3.3.1 Response

class Response(*content=''*, *status_code=STATUS.OK*, *content_type='text/html'*)

encoding

status_message

headers
A dict of HTTP headers

cookies
A SimpleCookie container for resposne cookies.

add_cookie(*key*, *value*, ***kwargs*)
Add a cookie to the response.

If only key and value are passed, then dict access to self.cookies is used. Otherwise, a Morsel is instanciated, and the key, value and kwargs passed to its set method. Then it's added to the cookies container.

status
A helper to return the status code and message as a single string.

Response sub-classes

Additionally, there is a sub-class of Response for each HTTP Status code.

3.3.2 STATUS_CODES

A tuple of two-tuples, each of (status code, status message)

3.3.3 STATUS

An OrderedDict sub-class constructed from STATUS_CODES.

Additionally, the status codes can be accessed by name.

For example:

```
>>> STATUS.OK
200

>>> STATUS[200]
'OK'
```

3.4 URL Dispatcher

Included is a Django-esque URL dispatcher view.

```
from functools import partial

from antfarm.views import urls

from myapp import views

view = urls.url_dispatcher(
    (r'^/$', views.index),
    (r'^/(?P<slug>[-\w]+)/$', views.blog_detail),
    (r'^/(?P<slug>[-\w]+)/up/$', partial(views.blog_vote, direction=1)),
    (r'^/(?P<slug>[-\w]+)/$', partial(views.blog_vote, direction=-1)),
)

application = App(root_view=view)
```

Note: Unlike Django, the initial / on the url is not automatically removed. To get a more django feel, you can include a pattern like this:

```
root_url = urls.url_dispatcher(
    (r'^/', root)
)
```

A view can raise a `antfarm.urls.KeepLooking` exception to tell the dispatcher to continue scanning.

3.4.1 urls_dispather.register

You can dynamically add patterns to a urls_dispatcher by calling the instances `register` method:

```
urls = url_dispatcher(....)

urls.register(pattern, view)
```

Additionally, you can decorate your views to add them to the url_dispatcher.

```
urls = url_dispatcher()

@urls.register(pattern)
def view(request...):
```

Nesting patterns

The currently unmatched portion of the path is stashed on the Request object as `remaining_path`, so `url_dispatcher` views can be nested.

```
inner_patterns = url_dispatcher(
    ...
)

root_view = url_dispatcher(
    ...
    (r'^/inner/', inner_patterns),
)
```

Custominsing Not Found

To control what response is given when no match is found for a pattern, you can sub-class url_dispatcher. Override `handle_not_found` method.

```
class my_url_dispatcher(url_dispatcher):
    def handle_not_found(self, request):
        return http.NotFound("Could not find a page for %s" % request.path)
```

3.5 Utilities

3.5.1 Functional

buffered_property

This works much like Python's `property` built-in, except it will only call the function once per instance, saving the result on the objects's __dict__.

In subsequent accesses to the property, Python will discover the value in __dict__ first, and skip calling the property's __get__.

In all other ways, this works as a normal class attribute. Setting and del work as expected.

By default, `buffered_property` will save the value to the name of the method it decorators. If you want to provide a buffered interface to a method, but keep the method, you will need to pass the name argument:

```
def get_foo(self):
    ...

foo = buffered_property(get_foo, name='foo')
```

3.6 Django equivalents

Documented here are antfarm equivalents to Django idioms.

3.6.1 Middleware

The need for middleware is obviated by the fact everything is a view. If you want to hook in something to do work before matching a view, before calling a view, or on the way out, you can just wrap that view in your own view.

This was a pattern proposed in Django also, to help disambiguate which middleware methods are called when, but it has not been included yet as it is too much of a backward-incompatible burden.

Further to this approach, it now becomes much simpler to selectively implement middleware, as you can wrap only the views or dispatcher paths you choose.

3.6.2 URL Patterns

There is a Django-style URL dispatcher view included in views/urls.py

There is currently no support for named url patterns or reversing urls.

3.7 Examples

A simple way to run any of these examples is with gunicorn:

```
gunicorn -b localhost:8000 test:application
```

3.7.1 Hello World!

```python
import antfarm

def index(request):
    return antfarm.Response('Hello World!')

application = antfarm.App(root_view=index)
```

3.7.2 Simple URL routing

```python
import antfarm
from antfarm.views.urls import url_dispatcher

def index(request):
    return antfarm.Response('Index')

def detail(request, user_pk):
    return antfarm.Response('You asked for %s' % user_pk)

application = antfarm.App(
    root_view = url_dispatcher(
        (r'^/$', index),
        (r'^/details/(?P<user_pk>\d+)/$', detail),
    )
)
```

3.8 Cookbook

Below are some common patterns that have proven productive in using Antfarm.

3.8.1 Middleware

It's easy to write "Middleware" style views, which do some work before or after other views.

```
class middleware(object):
    def __init__(self, view):
        self.view

    def __call__(self, request, *args, **kwargs):
        # Work before
        try:
            return self.view(request, *args, **kwargs)
        except ...:
            # Catch errors
        finally:
            # Work after _always_


 application = App(root_view = middleware(normalview))
```

3.8.2 Selective Middleware

An idea which resurfaces frequently in the Django community is one of applying middleware to a sub-set of the URL tree. The only existing solution is to apply a decorator to all the views [tedious and error prone] or to complicate the middleware with ways to denote what it is to apply to.

In Antfarm, this problem is trivially solved, since middleware are just views which wrap views.

A simple example is making some URLs password protected, but not others.

```
private_urls = url_dispatcher(
    (r'^$', views.user_list),
    (r'^(?P<user_id>\d+)/$', views.user_detail),
)

root_urls = url_dispatcher(
    (r'^/$', views.index),
    (r'^/login/$', views.login),
    (r'^/users/', login_required(private_urls)),
)
```

3.9 Testing

3.9.1 Running tests

A test suite using the standard library's `unittest` package exists in the `tests` directory of the git repository, it can be run from the root of the repository via:

```
python tests
```

To run only the tests in a specific file, you may do:

```
python tests/<filename>.py
```

To generate a coverage report for the test suite:

```
coverage run tests/__main__.py
```

Once the coverage data is generated, you can report on it using your preferred output method.

3.9.2 Writing tests

New tests should either be added to the appropriate test file, if it already exists, or to a new file in the `tests` directory, whose name is prefixed with `test_`:

```
tests/test_<thing_to_test>.py
```

Test **classes** should be written such that they subclass `unittest.TestCase` and are named with a `Test` suffix:

```
from unittest import TestCase

class ThingTest(TestCase):
    pass
```

Individual test **methods** should be named and numbered like so:

```
class ThingTest(TestCase):
    def test_001_function_description():
        pass

    def test_002_another_function_description():
        pass
```

Finally, to allow individual test files to be without the rest of the suite, the file should end with the following `if` statement:

```
from unittest import main

if __name__ == '__main__':
    main()
```

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*