Pando Documentation

Release 0.47-dev

Chad Whitacre et al.

Mar 24, 2020

Contents

This is Pando, a Python web framework.

Pando's source code is on GitHub, and is MIT-licensed.

CHAPTER 1

Installation

pando is available on PyPI:

```
$ pip install pando
```

CHAPTER 2

Contents

2.1 Tutorial

2.1.1 Quick Start

Given: POSIX and virtualenv

Step 1: Make a sandbox:

```
$ virtualenv foo
$ cd foo
$ . bin/activate
```

Step 2: Install pando from PyPI:

```
(foo)$ pip install pando
blah
blah
blah
```

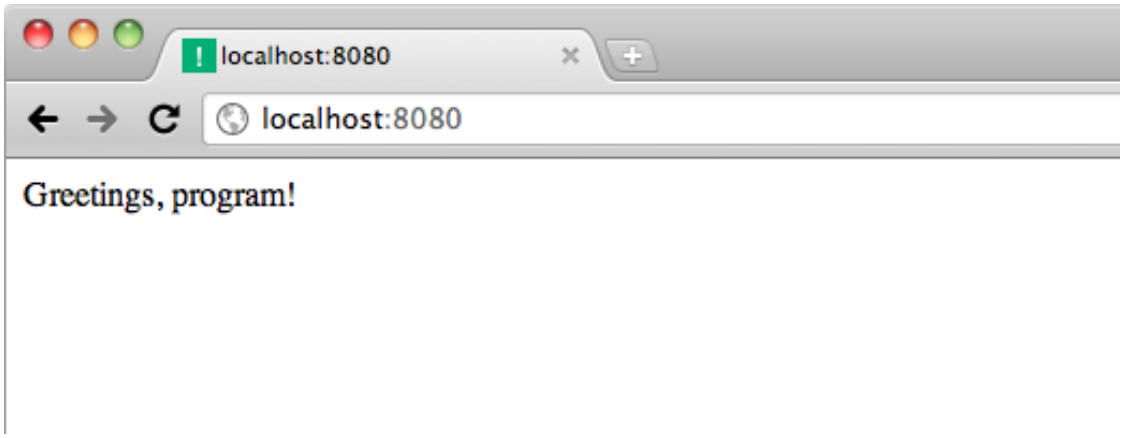
Step 3: Create a website root:

```
(foo)$ mkdir www
(foo)$ cd www
```

Step 4: Create a web page, and start pando inside it:

```
(foo)$ echo Greetings, program! > index.html.spt
(foo)$ python -m pando
Greetings, program! Welcome to port 8080.
```

Step 5: Check localhost for your new page!



2.2 Reference

This is the API reference for the Pando library.

2.2.1 body_parsers

This module contains Pando's built-in body parsers.

Body parsers are optional ways to enable Pando to uniformly parse POST body content according to its supplied `Content-Type`.

A body parser has the signature:

```
def name(raw, headers):
```

where `raw` is the raw bytestring to be parsed, and `headers` is the `Headers` mapping of the supplied headers.

`pando.body_parsers.`**`formdata`**(*raw*, *headers*)
Parse `raw` as form data.

Supports `application/x-www-form-urlencoded` and `multipart/form-data`.

`pando.body_parsers.`**`jsondata`**(*raw*, *headers*)
Parse `raw` as JSON data.

2.2.2 exceptions

Custom exceptions raised by Pando

exception `pando.exceptions.`**`CRLFInjection`**
A 400 *Response* (per #249) raised if there's a suspected CRLF Injection attack in the headers.

exception `pando.exceptions.`**`MalformedHeader`**(*header*)
A 400 *Response* (per RFC7230 section 3.2.4) raised if there's no `:` in a header field, or if there's leading or trailing whitespace in the key part of a header field.

exception `pando.exceptions.`**`MalformedBody`**(*msg*)
A 400 *Response* raised if parsing the body of a POST request fails.

exception pando.exceptions.**UnknownBodyType**(*ctype*)
A 415 *Response* raised if the `Content-Type` of the body of a POST request doesn't have a `body_parser` registered for it.

exception pando.exceptions.**BadLocation**(*msg*)
A 500 *Response* raised if an invalid redirect is attempted.

2.2.3 http

baseheaders

class pando.http.baseheaders.**BaseHeaders**(*d*)
Bases: *pando.http.mapping.CaseInsensitiveMapping*

Represent the headers in an HTTP Request or Response message.

How to send non-English unicode string using HTTP header? and What character encoding should I use for a HTTP header? have good notes on why we do everything as pure bytes here.

__init__(*d*)
Takes headers as a dict, list, or bytestring.

__setitem__(*name*, *value*)
Checks for CRLF in `value`, then calls the superclass method:

`CaseInsensitiveMapping.__setitem__`(*name*, *value*)

add(*name*, *value*)
Checks for CRLF in `value`, then calls the superclass method:

`CaseInsensitiveMapping.add`(*name*, *value*)

raw
Return the headers as a bytestring, formatted for an HTTP message.

__contains__(*k*) → True if D has a key k, else False

__getitem__(*name*)

all(*name*)

get(*name*, *default=None*)

keyerror(*name*)
Raises a 400 *Response*.

ones(**names*)
Given one or more names of keys, return a list of their values.

pop(*name*)

popall(*name*)
D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

mapping

class pando.http.mapping.**Mapping**
Bases: `aspen.http.mapping.Mapping`

keyerror(*name*)
Raises a 400 *Response*.

__getitem__(*name*)
Given a name, return the last value or call self.keyerror.

__setitem__(*name*, *value*)
Given a name and value, clobber any existing values.

add(*name*, *value*)
Given a name and value, clobber any existing values with the new one.

all(*name*)
Given a name, return a list of values, possibly empty.

get(*name*, *default=None*)
Override to only return the last value.

ones(**names*)
Given one or more names of keys, return a list of their values.

pop(*name*, *default=<object object>*)
Given a name, return a value.

This removes the last value from the list for name and returns it. If there was only one value in the list then the key is removed from the mapping. If name is not present and default is given, that is returned instead. Otherwise, self.keyerror is called.

popall()
D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

class pando.http.mapping.**CaseInsensitiveMapping**(**a*, ***kw*)
Bases: *pando.http.mapping.Mapping*

__init__(**a*, ***kw*)
Initializes the mapping.

Loops through positional arguments first, then through keyword args.

Positional arguments can be dicts or lists of items.

__contains__(*k*) → True if D has a key k, else False

__getitem__(*name*)

__setitem__(*name*, *value*)

add(*name*, *value*)

get(*name*, *default=None*)

all(*name*)

pop(*name*)

popall(*name*)
D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

keyerror(*name*)
Raises a 400 *Response*.

ones(**names*)
Given one or more names of keys, return a list of their values.

request

Define a Request class and child classes.

Here is how we analyze the structure of an HTTP message, along with the objects we use to model each:

```
- request                   Request
    - line                  Line
        - method            Method      ASCII
        - uri               URI
            - path          Path
              - parts       list of PathPart
            - querystring   Querystring
        - version           Version     ASCII
    - headers               Headers     str
        - cookie            Cookie      str
    - body                  Body        Content-Type?
```

pando.http.request.**make_franken_uri**(*path*, *qs*)

Given two bytestrings, return a bytestring.

We want to pass ASCII to Request. However, our friendly neighborhood WSGI servers do friendly neighborhood things with the Request-URI to compute PATH_INFO and QUERY_STRING. In addition, our friendly neighborhood browser sends "raw, unescaped UTF-8 bytes in the query during an HTTP request" (http://web.lookout.net/2012/03/unicode-normalization-in-urls.html).

Our strategy is to try decoding to ASCII, and if that fails (we don't have ASCII) then we'll quote the value before passing to Request. What encoding are those bytes? Good question. The above blog post claims that experiment reveals all browsers to send UTF-8, so let's go with that? BUT WHAT ABOUT MAXTHON?!?!?!.

pando.http.request.**make_franken_headers**(*environ*)

Takes a WSGI environ, returns a dict of HTTP headers.

https://www.python.org/dev/peps/pep-3333/#environ-variables

pando.http.request.**kick_against_goad**(*environ*)

Kick against the goad. Try to squeeze blood from a stone. Do our best.

class pando.http.request.**Request**(*website*, *method='GET'*, *uri='/'*, *server_software=''*, *version='HTTP/1.1'*, *headers=''*, *body=None*)

Bases: object

Represent an HTTP Request message.

line

See *Line*.

headers

A mapping of HTTP headers. See Headers.

__init__(*website*, *method='GET'*, *uri='/'*, *server_software=''*, *version='HTTP/1.1'*, *headers=''*, *body=None*)

body is expected to be a file-like object.

classmethod from_wsgi(*website*, *environ*)

Given a WSGI environ, return a new instance of the class.

The conversion from HTTP to WSGI is lossy. This method does its best to go the other direction, but we can't guarantee that we've reconstructed the bytes as they were on the wire.

Almost all the keys and values in a WSGI environ dict are (supposed to be) of type *str*, meaning bytestrings in python 2 and unicode strings in python 3. In this function we normalize them to bytestrings. Ref:

https://www.python.org/dev/peps/pep-3333/#a-note-on-string-types

method

path

qs

cookie

content_length
This property attempts to parse the `Content-Length` header.

Returns zero if the header is missing or empty.

Raises a 400 *Response* if the header is not a valid integer.

body_bytes
Lazily read the whole request body.

Returns `b''` if the request doesn't have a body.

body
This property calls *parse_body()* and caches the result.

parse_body()
Parses *body_bytes* using *headers* to determine which of the *body_parsers* should be used.

Raises *UnknownBodyType* if the HTTP `Content-Type` isn't recognized, and *MalformedBody* if the parsing fails.

host
The hostname of the request.

Raises a 400 *Response* if no `Host` header is found or if decoding it fails. See RFC7230 section 5.4.

scheme
The guessed URL scheme of the request, usually 'https' or 'http'.

If the `website.trusted_proxies` list is empty, then the value of the WSGI variable `url_scheme` is returned, otherwise the value of the X-Forwarded-Proto HTTP header is returned.

Support for RFC7239 may be added in the future (patches welcome ;-)).

If the scheme cannot be determined or isn't in *known_schemes*, then a `Warning` is emitted and 'https' is returned, because it's better to fail safely than to downgrade to an insecure connection.

source
The IP address of the client (an `IPv4Address` or `IPv6Address` object).

This property looks at WSGI's `REMOTE_ADDR` variable and HTTP's `X-Forwarded-For` header.

> **Warning:** Make sure to correctly fill the *trusted_proxies* list, otherwise this property will return the IP address of the reverse proxy.

bypasses_proxy
This property returns `False` if the request came through all the proxy levels listed in *trusted_proxies*, and `True` if the request bypassed at least one proxy level.

__str__()
Lazily load the body and return the whole message.

When working with a Request object interactively or in a debugging situation we want it to behave transparently string-like. We don't want to read bytes off the wire if we can avoid it, though, because for mega file uploads and such this could have a big impact.

__repr__(*)* <==> *repr(x)*

__cmp__(*other*)

allow(**methods*)
Given method strings, raise 405 if ours is not among them.

The method names are case insensitive (they are uppercased). If 405 is raised then the Allow header is set to the methods given.

is_xhr()
Check the value of X-Requested-With.

class pando.http.request.**Line**
Bases: str

Represent the first line of an HTTP Request message.

static __new__(*cls*, *method*, *uri*, *version*)
Takes three bytestrings.

pando.http.request.**STANDARD_METHODS = set([u'CONNECT', u'DELETE', u'GET', u'HEAD', u'OPTIO**
A set containing the 8 basic HTTP methods.

If your application uses other standard methods (see the HTTP Method Registry), or custom methods, you can add them to this set to improve performance.

class pando.http.request.**Method**
Bases: str

Represent the HTTP method in the first line of an HTTP Request message.

static __new__(*cls*, *raw*)
Creates a new Method object.

Raises a 400 *Response* if the given bytestring is not a valid HTTP method, per RFC7230 section 3.1.1:

> Recipients of an invalid request-line SHOULD respond with either a 400 (Bad Request) error or a 301 (Moved Permanently) redirect with the request-target properly encoded.

RFC7230 defines valid methods as:

```
method         = token

token          = 1*tchar

tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
               / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
               / DIGIT / ALPHA
               ; any VCHAR, except delimiters
```

class pando.http.request.**URI**
Bases: str

Represent the Request-URI in the first line of an HTTP Request message.

static __new__(*cls*, *raw*)
Creates a URI object from a raw bytestring.

We require that `raw` be decodable with ASCII, if it isn't a 400 `Response` is raised.

class pando.http.request.**Path**
Bases: str

decoded
The path decoded to text.

mapping
Mapping of path variables.

parts
List of PathPart instances.

static __new__(*cls*, *raw*)
Creates a Path object from a raw bytestring.

class pando.http.request.**Querystring**
Bases: str

decoded
The querystring decoded to text.

mapping
Mapping of querystring variables.

static __new__(*cls*, *raw*)
Creates a Querystring object from a raw bytestring.

class pando.http.request.**Version**
Bases: str

Holds the version from the HTTP status line, e.g. HTTP/1.1.

Accessing the *info*, *major*, or *minor* attribute will raise a 400 *Response* if the version is invalid.

RFC7230 section 2.6:

```
HTTP-version  = HTTP-name "/" DIGIT "." DIGIT
HTTP-name     = %x48.54.54.50 ; "HTTP", case-sensitive
```

__slots__ = []

info

major

minor

safe_decode()

response

class pando.http.response.**CloseWrapper**(*request*, *body*)
Conform to WSGI's facility for running code *after* a response is sent.

__iter__()

close()

exception pando.http.response.**Response**(*code=200*, *body=u''*, *headers=None*)
Represent an HTTP Response message.

request = None

whence_raised = (None, None)

__init__(*code=200*, *body=u''*, *headers=None*)
Takes an int, a string, a dict.

- code an HTTP response code, e.g., 404
- body the message body as a string
- headers a dict, list, or bytestring of HTTP headers

Code is first because when you're raising your own Responses, they're usually error conditions. Body is second because one more often wants to specify a body without headers, than a header without a body.

to_wsgi(*environ*, *start_response*, *charset*)

__repr__(*x*) <==> *repr(x)*

__str__(*x*) <==> *str(x)*

set_whence_raised()
Sets self.whence_raised

It's a tuple, (filename, linenum) where we were raised from.

This function needs to be called from inside the *except* block.

2.2.4 logging

Pando logging convenience wrappers

pando.logging.**log**(**messages*, ***kw*)
Make logging more convenient - use magic to get the __name__ of the calling module/function and log as it.

'level' if present as a kwarg, is the level to log at. 'upframes' if present as a kwarg, is how many frames up to look for the name.

other kwargs are passed through to Logger.log()

pando.logging.**log_dammit**(**messages*, ***kw*)
like log(), but critical instead of warning

2.2.5 state_chain

These functions comprise the request processing functionality of Pando.

The order of functions in this module defines Pando's state chain for request processing. The actual parsing is done by StateChain.from_dotted_name().

Dependencies are injected as specified in each function definition. Each function should return `None`, or a dictionary that will be used to update the state in the calling routine.

It's important that function names remain relatively stable over time, as downstream applications are expected to insert their own functions into this chain based on the names of our functions here. A change in function names or ordering here would constitute a backwards-incompatible change.

pando.state_chain.**parse_environ_into_request**(*environ*, *website*)

pando.state_chain.**request_available**()
No-op placeholder for easy hookage

pando.state_chain.**raise_200_for_OPTIONS**(*request*)
A hook to return 200 to an 'OPTIONS *' request

pando.state_chain.**redirect_to_base_url**(*website*, *request*)

pando.state_chain.**dispatch_path_to_filesystem**(*website*, *request*)

pando.state_chain.**raise_404_if_missing**(*dispatch_result*, *website*)

pando.state_chain.**redirect_to_canonical_path**(*dispatch_result*, *website*)

pando.state_chain.**apply_typecasters_to_path**(*state*, *website*, *request*)

pando.state_chain.**load_resource_from_filesystem**(*website*, *dispatch_result*)

pando.state_chain.**resource_available**()
No-op placeholder for easy hookage

pando.state_chain.**create_response_object**(*state*)

pando.state_chain.**extract_accept_header**(*request=None*, *exception=None*)

pando.state_chain.**render_response**(*state*, *resource*, *response*, *website*)

pando.state_chain.**handle_negotiation_exception**(*exception*)

pando.state_chain.**get_response_for_exception**(*website*, *exception*)

pando.state_chain.**response_available**()
No-op placeholder for easy hookage

pando.state_chain.**log_traceback_for_5xx**(*response*, *traceback=None*)

pando.state_chain.**delegate_error_to_simplate**(*website*, *state*, *response*, *request=None*, *resource=None*)

pando.state_chain.**log_traceback_for_exception**(*website*, *exception*)

pando.state_chain.**log_result_of_request**(*website*, *request=None*, *dispatch_result=None*, *response=None*)
Log access. With our own format (not Apache's).

2.2.6 testing

client

exception pando.testing.client.**DidntRaiseResponse**

class pando.testing.client.**FileUpload**(*data*, *filename*, *content_type=None*)
Model a file upload for testing. Takes data and a filename.

pando.testing.client.**encode_multipart**(*boundary*, *data*)
Encodes multipart POST data from a dictionary of form values.

The key will be used as the form data name; the value will be transmitted as content. Use the FileUpload class to simulate file uploads (note that they still come out as FieldStorage instances inside of simplates).

class pando.testing.client.**Client**(*www_root=None*, *project_root=None*)
This is the Pando test client. It is probably useful to you.

hydrate_website(***kwargs*)

website

load_resource(*path*)
Given an URL path, return a Resource instance.

get_session()

GET(**a*, ***kw*)

POST(**a*, ***kw*)

OPTIONS(**a*, ***kw*)

HEAD(**a*, ***kw*)

PUT(**a*, ***kw*)

DELETE(**a*, ***kw*)

TRACE(**a*, ***kw*)

CONNECT(**a*, ***kw*)

GxT(**a*, ***kw*)

PxST(**a*, ***kw*)

xPTIONS(**a*, ***kw*)

HxAD(**a*, ***kw*)

PxT(**a*, ***kw*)

DxLETE(**a*, ***kw*)

TRxCE(**a*, ***kw*)

CxNNECT(**a*, ***kw*)

hxt(**a*, ***kw*)

hit(*method*, *path=u'/'*, *data=None*, *body=''*, *content_type='multipart/form-data; boundary=BoUnDaRyStRiNg'*, *raise_immediately=True*, *return_after=None*, *want=u'response'*, ***headers*)

static resolve_want(*state*, *want*)

build_wsgi_environ(*method*, *url*, *body*, *content_type*, *cookies=None*, ***kw*)

class pando.testing.client.**StatefulClient**(**a*, ***kw*)

This is a Client subclass that keeps cookies between calls.

__enter__()

__exit__(**a*)

hit(**a*, ***kw*)

harness

pando.testing.harness.**teardown**()

Standard teardown function.

- reset the current working directory
- remove FSFIX = %{tempdir}/fsfix
- clear out sys.path_importer_cache

class pando.testing.harness.**Harness**

A harness to be used in the Pando test suite itself. Probably not useful to you.

teardown()

simple(*contents=u'Greetings*, *program!'*, *filepath=u'index.html.spt'*, *uripath=None*, *website_configuration=None*, ***kw*)

A helper to create a file and hit it through our machinery.

make_request(*a, **kw)

make_dispatch_result(*a, **kw)

2.2.7 utils

pando.utils.**maybe_encode**(*s*, *codec=u'ascii'*)

pando.utils.**total_seconds**(*td*)
Python 2.7 adds a total_seconds method to timedelta objects.

See http://docs.python.org/library/datetime.html#datetime.timedelta.total_seconds

This function is taken from https://bitbucket.org/jaraco/jaraco.compat/src/e5806e6c1bcb/py26compat/__init__.py#cl-26

class pando.utils.**UTC**
UTC - http://docs.python.org/library/datetime.html#tzinfo-objects

utcoffset(*dt*)
datetime -> minutes east of UTC (negative for west of UTC).

tzname(*dt*)
datetime -> string name of time zone.

dst(*dt*)
datetime -> DST offset in minutes east of UTC.

pando.utils.**utcnow**()
Return a tz-aware datetime.datetime.

pando.utils.**to_rfc822**(*dt*)
Given a datetime.datetime, return an RFC 822-formatted unicode.

Sun, 06 Nov 1994 08:49:37 GMT

According to RFC 1123, day and month names must always be in English. If not for that, this code could use strftime(). It can't because strftime() honors the locale and could generated non-English names.

pando.utils.**typecheck**(**checks*)
Assert that arguments are of a certain type.

Checks is a flattened sequence of objects and target types, like this:

```
( {'foo': 2}, dict
, [1,2,3], list
, 4, int
, True, bool
, 'foo', (basestring, None)
 )
```

The target type can be a single type or a tuple of types. None is special-cased (you can specify None and it will be interpreted as type(None)).

```
>>> typecheck()
>>> typecheck('foo')
Traceback (most recent call last):
    ...
AssertionError: typecheck takes an even number of arguments.
>>> typecheck({'foo': 2}, dict)
>>> typecheck([1,2,3], list)
```

(continues on next page)

(continued from previous page)

```
>>> typecheck(4, int)
>>> typecheck(True, bool)
>>> typecheck('foo', (str, None))
>>> typecheck(None, None)
>>> typecheck(None, type(None))
>>> typecheck('foo', unicode)
Traceback (most recent call last):
    ...
TypeError: Check #1: 'foo' is of type str, but unicode was expected.
>>> typecheck('foo', (basestring, None))
Traceback (most recent call last):
    ...
TypeError: Check #1: 'foo' is of type str, not one of: basestring, NoneType.
>>> class Foo(object):
...   def __repr__(self):
...     return "<Foo>"
...
>>> typecheck(Foo(), dict)
Traceback (most recent call last):
    ...
TypeError: Check #1: <Foo> is of type __main__.Foo, but dict was expected.
>>> class Bar:
...   def __repr__(self):
...     return "<Bar>"
...
>>> typecheck(Bar(), dict)
Traceback (most recent call last):
    ...
TypeError: Check #1: <Bar> is of type instance, but dict was expected.
>>> typecheck('foo', str, 'bar', unicode)
Traceback (most recent call last):
    ...
TypeError: Check #2: 'bar' is of type str, but unicode was expected.
```

2.2.8 website

class pando.website.**Website**(***kwargs*)

Represent a website.

This object holds configuration information, and how to handle HTTP requests (per WSGI). It is available to user-developers inside of their simplates and state chain functions.

Parameters **kwargs** – configuration values. The available options and their default values are described in *pando.website.DefaultConfiguration* and aspen.request_processor.DefaultConfiguration.

request_processor = None

An Aspen RequestProcessor instance.

state_chain = None

The chain of functions used to process an HTTP request, imported from *pando.state_chain*.

body_parsers = None

Mapping of content types to parsing functions.

__call__(*environ*, *start_response*)

Alias of *wsgi_app()*.

wsgi_app(*environ*, *start_response*)
WSGI interface.

Wrap this method (instead of the website object itself) when you want to use WSGI middleware:

```
website = Website()
website.wsgi_app = WSGIMiddleware(website.wsgi_app)
```

respond(*environ*, *raise_immediately=None*, *return_after=None*)
Given a WSGI environ, return a state dict.

redirect(*location*, *code=None*, *permanent=False*, *base_url=None*, *response=None*)
Raise a redirect Response.

If code is None then it will be set to 301 (Moved Permanently) if permanent is True and 302 (Found) if it is False. If url doesn't start with base_url (defaulting to self.base_url), then we prefix it with base_url before redirecting. This is a protection against open redirects. If you wish to use a relative path or full URL as location, then base_url must be the empty string; if it's not, we raise BadLocation. If you provide your own response we will set .code and .headers['Location'] on it.

canonicalize_base_url(*request*)
Enforces a base_url such as http://localhost:8080 (no path part).

See *Request.host* and *Request.scheme* for how the request host and scheme are determined.

find_ours(*filename*)
Given a `filename`, return the filepath to pando's internal version of that filename.

No existence checking is done, this just abstracts away the `__file__` reference nastiness.

ours_or_theirs(*filename*)
Given a filename, return a filepath or `None`.

It looks for the file in `self.project_root`, then in Pando's default files directory. `None` is returned if the file is not found in either location.

default_renderers_by_media_type
Reference to `Simplate.default_renderers_by_media_type`, for backward compatibility.

project_root
Reference to `self.request_processor.project_root`, for backward compatibility.

renderer_factories
Reference to `Simplate.renderer_factories`, for backward compatibility.

www_root
Reference to `self.request_processor.www_root`, for backward compatibility.

class pando.website.**DefaultConfiguration**
Default configuration of *Website* objects.

base_url = u''
The website's base URL (scheme and host only, no path). If specified, then requests for URLs that don't match it are automatically redirected. For example, if `base_url` is `https://example.net`, then a request for `http://www.example.net/foo` is redirected to `https://example.net/foo`.

colorize_tracebacks = True
Use the Pygments package to prettify tracebacks with syntax highlighting.

known_schemes = set([u'http', u'https', u'ws', u'wss'])
The set of known and acceptable request URL schemes. Used by *Request.scheme*.

list_directories = False
List the contents of directories that don't have a custom index.

show_tracebacks = False
Show Python tracebacks in error responses.

trusted_proxies = []
The list of reverse proxies that requests to this website go through. With this information we can accurately determine where a request came from (i.e. the IP address of the client) and how it was sent (i.e. encrypted or in plain text).

Example:

```
trusted_proxies=[
    ['private'],
    [IPv4Network('1.2.3.4/32'), IPv6Network('2001:2345:6789:abcd::/64')]
]
```

Explanation: *trusted_proxies* is a list of proxy levels, with each item being a list of IP networks (IPv4Network or IPv6Network objects). The special value 'private' can be used to indicate that any private IP address is trusted (the is_private attribute is used to determine if an IP address is private, both IPv4 and IPv6 are supported).

2.2.9 wsgi

Provide a WSGI callable.

(It could be nice if this was at pando:wsgi instead of pando.wsgi:website, but then Website would be instantiated every time you import the pando module. Here, it's only instantiated when you pass this to a WSGI server like gunicorn, spawning, etc.)

pando.wsgi.**website = <pando.website.Website object>**
This is the WSGI callable, an instance of *Website*.

pando.wsgi.**application = <pando.website.Website object>**
Alias of website. A number of WSGI servers look for this name by default, for example running gunicorn pando.wsgi works.

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

S

T

U

V

W

X