AdvancedHTTPServer Documentation

Release 2.2.0

Spencer McIntyre

Mar 22, 2019

Technical Documentation

AdvancedHTTPServer is a light weight module that provides a set of classes for quickly making HTTP servers for a variety of purposes. It focuses on a light and powerful design with an emphasis on portability. It was designed after and builds upon Python's standard `http.server` module module. AdvancedHTTPServer is released under the BSD license and can be freely distributed and packaged with other software.

CHAPTER 1

Features

AdvancedHTTPServer provides out of the box support for additional commonly needed features such as:

- Threaded request handling
- Binding to multiple interfaces
- SSL and SNI support
- Registering handler functions to HTTP resources
- A default robots.txt file
- Basic authentication
- The HTTP verbs GET, HEAD, POST, and OPTIONS
- Remote Procedure Call (RPC) over HTTP
- WebSockets

1.1 Getting Started

The AdvancedHTTPServer module is composed of two main classes which implement the bulk of the provided functionality. These two classes are `AdvancedHTTPServer` and `RequestHandler`. Just like Python's `http.server` module, the server takes a class **not an instance of a class** and is responsible for responding to individual requests at the TCP connection level. The `RequestHandler` instance is initialized automatically by the server when a request is received.

The following sections outline how to accomplish common tasks using AdvancedHTTPServer.

1.1.1 Binding To Interfaces

Bind to a single interface using the *address* (singular) keyword argument to the `AdvancedHTTPServer.__init__()` method.

```
server = AdvancedHTTPServer(RequestHandler, address=('0.0.0.0', 8081))
```

Deprecated since version 2.0.12: The *address* keyword argument has been deprecated in favor of the *addresses* keyword argument. It should not be used in new code.

Bind to one or more interfaces using the *addresses* (plural) keyword argument to the `AdvancedHTTPServer.__init__()` method.

```
server = AdvancedHTTPServer(RequestHandler, addresses=(
    # address,  port,  use ssl
    ('0.0.0.0', 80,    False),
    ('0.0.0.0', 8080,  False)
))
```

1.1.2 Enabling SSL

To enable SSL, pass a PEM file path using the *ssl_certfile* keyword argument to the `AdvancedHTTPServer.__init__()` method. This will be the default certificate. Additional certificates can be configured with TLS's Server Name Indication (SNI) extension using the `AdvancedHTTPServer.add_sni_cert()` method.

```
server = AdvancedHTTPServer(RequestHandler,
    address=('0.0.0.0', 443),
    ssl_certfile='/path/to/the/certificate.pem'
)
```

An insecure, self-signed certificate suitable for testing can be created using the following openssl command:

```
openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
```

1.1.3 Enabling Basic Authentication

Basic authentication can be enabled by adding credentials to a `AdvancedHTTPServer` instance using its `AdvancedHTTPServer.auth_add_creds()` method which takes a username and password. The *pwtype* keyword argument can optionally be used to specify that the password is a hash.

```
server = AdvancedHTTPServer(RequestHandler)
server.auth_add_creds('admin', 'Sup3rS3cr3t!')
```

1.1.4 Using RPC

AdvancedHTTPServer supports a custom form of RPC over HTTP using the `RPC` verb. To register RPC methods in a `RequestHandler` they must be added to the `RequestHandler.rpc_handler_map` dictionary. Unlike standard HTTP request handlers, RPC request handlers can take arbitrary arguments and key word arguments.

To define an RPC capable `RequestHandler`:

```
# define a custom RequestHandler inheriting from the original
class RPCHandler(RequestHandler):
    def on_init(self):
        # add to rpc_handler_map instead of handler_map
        self.rpc_handler_map['/xor'] = self.rpc_xor
```

(continues on next page)

(continued from previous page)

```
    def rpc_xor(self, key, data):
        return ''.join(map(lambda x: chr(ord(x) ^ key), data))

# initialize the server with the custom handler
server = AdvancedHTTPServer(RPCHandler)
```

To call methods from an RPC capable *RequestHandler*:

```
# in this case the server is running at http://localhost:8080/
rpc = RPCClient(('localhost', 8080))
rpc('xor', 1, 'test')
```

1.1.5 Passing Variables To The Request Handler

The *RequestHandler* instance is passed the instance of the *ServerNonThreaded* which received the request. This attribute can be used to pass forward values from the top level *AdvancedHTTPServer* object.

```
class DemoHandler(RequestHandler):
    def do_init(self):
        # access the value from the subserver instance
        self.some_value = self.server.some_value

class DemoServer(AdvancedHTTPServer):
    def __init__(self, some_value, *args, **kwargs):
        # initialize the server first, this sets self.sub_servers
        super(DemoServer, self).__init__(*args, **kwargs)
        # iterate through self.sub_servers and set the attribute to forward
        for server in self.sub_servers:
            server.some_value = some_value

some_value = 'Hello World!'
server = DemoServer(some_value, DemoHandler)
```

1.1.6 Registering Request Handlers

AdvancedHTTPServer provides two distinct methods of registering methods to handle either HTTP or RPC requests. These methods are provided so the user may select the one they prefer to work with.

Modifying The Handler Map

The *RequestHandler* class initializes the empty dictionaries for *RequestHandler.handler_map* and *RequestHandler.rpc_handler_map*. Both are keyed by a regular expression which is applied to the path of the HTTP request to find a valid handler method. These maps can be set by overriding the *RequestHandler.on_init()* method hook. The method must take a single argument (in addition to the standard class method `self` argument which goes first) which is the parsed query string.

```
class DemoHandler(RequestHandler):
    def on_init(self):
        # over ride on_init and add a generic http request handler method
        # this references a method which is defined later
        self.handler_map['^hello-world$'] = self.res_hello_world
```

(continues on next page)

(continued from previous page)

```
    def res_hello_world(self, query):
        # ...
        return
```

Using RegisterPath

The *RegisterPath* class can be used as a decorator to allow handler methods to be registered in the handler map. This approach does not require writing a *RequestHandler* class and the handlers can be simple functions. The functions must take two arguments, the first is the active *RequestHandler* instance and the second is the parsed query string.

The *handler* keyword argument to *RegisterPath.__init__()* specifies an optional *RequestHandler* to register the handler method with. By default, the handler is treated as a global handler and is registered for all *RequestHandler* instances. Alternatively, a specific handler can be specified either by a reference to the class or by the class's name.

```
# register a global handler for all RequestHandler instances
@RegisterPath('^register-path-global$')
def register_path_global(server, query):
    # ...
    return

# register a handler only for DemoHandler by it's name
@RegisterPath('^register-path-name$', 'DemoHandler')
def register_path_name(server, query):
    # ...
    return
# register a handler only for DemoHandler by it's class reference
@RegisterPath('^register-path-class$', DemoHandler)
def register_path_class(server, query):
    # ...
    return
```

Stacking RegisterPath

Since *RegisterPath* does not modify or wrap the handler method it is possible to "stack" the decorators to register a single handler for multiple paths.

```
@RegisterPath('^register-path-class-double$', DemoHandler)
@RegisterPath('^register-path-class$', DemoHandler)
def register_path_class(server, query):
    # ...
    return
```

1.1.7 Handling Requests

HTTP requests (and RPC requests) are dispatched to handlers defined by the *RequestHandler*. Two dictionaries exist, one for dispatching HTTP requests and another specifically for RPC requests. Both dictionaries use regular expressions as keys and functions to be called as value.

Standard HTTP requests such as GET and POST use the following standard function signature:

```
def some_http_handler(self, query):
    message = b'Hello World!\r\n\r\n'
    self.send_response(200)
    self.send_header('Content-Type', 'text/plain')
    self.send_header('Content-Length', len(message))
    self.end_headers()
    self.wfile.write(message)
    return
```

RPC requests use an arbitrary function signature supporting both positional (required) and keyword (optional) arguments. The caller must then specify these arguments as necessary following the standard Python rules. The value returned by an RPC handler is returned to the remote caller.

```
# define an RPC handler method accepting two arguments
def some_rpc_handler(self, arg1, kwarg1=None):
    # return None to the caller
    return
```

Accessing Headers

Request headers can be accessed from both standard HTTP and RPC handlers through the `RequestHandler.headers` attribute. Header strings are **case insensitive**.

```
def some_http_handler(self, query):
    # get the Accept header if it exists, otherwise an empty string
    accept_header = self.headers.get('Accept', '')
    message = b'Accept Header: ' + accept_header.encode('utf-8')
    self.send_response(200)
    self.send_header('Content-Type', 'text/plain')
    self.send_header('Content-Length', len(message))
    self.end_headers()
    self.wfile.write(message)
    return
```

Accessing Query Parameters

HTTP requests are passed the parsed query parameters in the *query* argument to the registered handler. This parameter is a dictionary keyed by the field name with a list of the values defined for the field name.

Note: The parsed query data uses an array for the value to store each occurrence of field. Usually it's desirable to just access the first or last instance but it is important to note that all are available.

```
def some_http_handler(self, query):
    # get the value of id from the query or a list containing an empty string
    # so the first member can be referenced without raising an exception
    id_value = query.get('id', [''])[0]
    message = b'id value: ' + id_value.encode('utf-8')
    self.send_response(200)
    self.send_header('Content-Type', 'text/plain')
    self.send_header('Content-Length', len(message))
    self.end_headers()
    self.wfile.write(message)
    return
```

1.2 `advancedhttpserver` – API Reference

1.2.1 Data

`g_serializer_drivers`

`g_ssl_has_server_sni`

1.2.2 Functions

`build_server_from_argparser`(*description=None*, *server_klass=None*, *handler_klass=None*)

Build a server from command line arguments. If a ServerClass or HandlerClass is specified, then the object must inherit from the corresponding AdvancedHTTPServer base class.

Parameters

- **`description`** (*`str`*) – Description string to be passed to the argument parser.
- **`server_klass`** (*`AdvancedHTTPServer`*) – Alternative server class to use.
- **`handler_klass`** (*`RequestHandler`*) – Alternative handler class to use.

Returns A configured server instance.

Return type *`AdvancedHTTPServer`*

`build_server_from_config`(*config*, *section_name*, *server_klass=None*, *handler_klass=None*)

Build a server from a provided `configparser.ConfigParser` instance. If a ServerClass or HandlerClass is specified, then the object must inherit from the corresponding AdvancedHTTPServer base class.

Parameters

- **`config`** (`configparser.ConfigParser`) – Configuration to retrieve settings from.
- **`section_name`** (*`str`*) – The section name of the configuration to use.
- **`server_klass`** (*`AdvancedHTTPServer`*) – Alternative server class to use.
- **`handler_klass`** (*`RequestHandler`*) – Alternative handler class to use.

Returns A configured server instance.

Return type *`AdvancedHTTPServer`*

`random_string`(*size*)

Generate a random string of *size* length consisting of both letters and numbers. This function is not meant for cryptographic purposes and should not be used to generate security tokens.

Parameters **`size`** (*`int`*) – The length of the string to return.

Returns A string consisting of random characters.

Return type str

`resolve_ssl_protocol_version`(*version=None*)

Look up an SSL protocol version by name. If *version* is not specified, then the strongest protocol available will be returned.

Parameters **`version`** (*`str`*) – The name of the version to look up.

Returns A protocol constant from the `ssl` module.

Return type int

1.2.3 Classes

class AdvancedHTTPServer(*handler_klass*, *address=None*, *addresses=None*, *use_threads=True*, *ssl_certfile=None*, *ssl_keyfile=None*, *ssl_version=None*)

This is the primary server class for the AdvancedHTTPServer module. Custom servers must inherit from this object to be compatible. When no *address* parameter is specified the address '0.0.0.0' is used and the port is guessed based on if the server is run as root or not and SSL is used.

__init__(*handler_klass*, *address=None*, *addresses=None*, *use_threads=True*, *ssl_certfile=None*, *ssl_keyfile=None*, *ssl_version=None*)

Parameters

- **handler_klass** (`RequestHandler`) – The request handler class to use.
- **address** (`tuple`) – The address to bind to in the format (host, port).
- **addresses** (`tuple`) – The addresses to bind to in the format (host, port, ssl).
- **use_threads** (`bool`) – Whether to enable the use of a threaded handler.
- **ssl_certfile** (`str`) – An SSL certificate file to use, setting this enables SSL.
- **ssl_keyfile** (`str`) – An SSL certificate file to use.
- **ssl_version** – The SSL protocol version to use.

add_sni_cert(*hostname*, *ssl_certfile=None*, *ssl_keyfile=None*, *ssl_version=None*)

Add an SSL certificate for a specific hostname as supported by SSL's Server Name Indicator (SNI) extension. See **RFC 3546** for more details on SSL extensions. In order to use this method, the server instance must have been initialized with at least one address configured for SSL.

Warning: This method will raise a `RuntimeError` if either the SNI extension is not available in the `ssl` module or if SSL was not enabled at initialization time through the use of arguments to *`__init__()`*.

New in version 2.0.0.

Parameters

- **hostname** (`str`) – The hostname for this configuration.
- **ssl_certfile** (`str`) – An SSL certificate file to use, setting this enables SSL.
- **ssl_keyfile** (`str`) – An SSL certificate file to use.
- **ssl_version** – The SSL protocol version to use.

auth_add_creds(*username*, *password*, *pwtype='plain'*)

Add a valid set of credentials to be accepted for authentication. Calling this function will automatically enable requiring authentication. Passwords can be provided in either plaintext or as a hash by specifying the hash type in the *pwtype* argument.

Parameters

- **username** (`str`) – The username of the credentials to be added.
- **password** (`bytes, str`) – The password data of the credentials to be added.
- **pwtype** (`str`) – The type of the *password* data, (plain, md5, sha1, etc.).

auth_delete_creds(*username=None*)

Delete the credentials for a specific username if specified or all stored credentials.

Parameters **`username`** (*`str`*) – The username of the credentials to delete.

`auth_set` (*status*)

Enable or disable requiring authentication on all incoming requests.

Parameters **`status`** (*`bool`*) – Whether to enable or disable requiring authentication.

`remove_sni_cert` (*hostname*)

Remove the SSL Server Name Indicator (SNI) certificate configuration for the specified *hostname*.

> **Warning:** This method will raise a `RuntimeError` if either the SNI extension is not available in the `ssl` module or if SSL was not enabled at initialization time through the use of arguments to *`__init__()`*.

New in version 2.2.0.

Parameters **`hostname`** (*`str`*) – The hostname to delete the SNI configuration for.

`serve_files`

Whether to enable serving files or not.

Type bool

`serve_files_list_directories`

Whether to list the contents of directories. This is only honored when *`serve_files`* is True.

Type bool

`serve_files_root`

The web root to use when serving files.

Type str

`serve_forever` (*fork=False*)

Start handling requests. This method must be called and does not return unless the *`shutdown()`* method is called from another thread.

Parameters **`fork`** (*`bool`*) – Whether to fork or not before serving content.

Returns The child processes PID if *fork* is set to True.

Return type int

`serve_robots_txt`

Whether to serve a default robots.txt file which denies everything.

Type bool

`server_started`

`server_version`

The server version to be sent to clients in headers.

Type str

`shutdown` ()

Shutdown the server and stop responding to requests.

`sni_certs`

New in version 2.2.0.

Returns Return a tuple of *`SSLSNICertificate`* instances for each of the certificates that are configured.

Return type tuple

sub_servers = None
The instances of *ServerNonThreaded* that are responsible for listening on each configured address.

class RegisterPath(*path*, *handler=None*, *is_rpc=False*)
Register a path and handler with the global handler map. This can be used as a decorator. If no handler is specified then the path and function will be registered with all *RequestHandler* instances.

```
@RegisterPath('^test$')
def handle_test(handler, query):
    pass
```

__init__(*path*, *handler=None*, *is_rpc=False*)

Parameters

- **path** (*str*) – The path regex to register the function to.
- **handler** (*str*) – A specific *RequestHandler* class to register the handler with.
- **is_rpc** (*bool*) – Whether the handler is an RPC handler or not.

class RequestHandler(**args*, ***kwargs*)
This is the primary http request handler class of the AdvancedHTTPServer framework. Custom request handlers must inherit from this object to be compatible. Instances of this class are created automatically. This class will handle standard HTTP GET, HEAD, OPTIONS, and POST requests. Callback functions called handlers can be registered to resource paths using regular expressions in the *handler_map* attribute for GET HEAD and POST requests and *rpc_handler_map* for RPC requests. Non-RPC handler functions that are not class methods of the request handler instance will be passed the instance of the request handler as the first argument.

basic_auth_user = None
The name of the user if the current request is using basic authentication.

check_authorization()
Check for the presence of a basic auth Authorization header and if the credentials contained within in are valid.

Returns Whether or not the credentials are valid.

Return type bool

cookie_get(*name*)
Check for a cookie value by name.

Parameters name (*str*) – Name of the cookie value to retreive.

Returns Returns the cookie value if it's set or None if it's not found.

cookie_set(*name*, *value*)
Set the value of a client cookie. This can only be called while headers can be sent.

Parameters

- **name** (*str*) – The name of the cookie value to set.
- **value** (*str*) – The value of the cookie to set.

dispatch_handler(*query=None*)
Dispatch functions based on the established handler_map. It is generally not necessary to override this function and doing so will prevent any handlers from being executed. This function is executed automatically when requests of either GET, HEAD, or POST are received.

Parameters query (*dict*) – Parsed query parameters from the corresponding request.

end_headers()
Send the blank line ending the MIME headers.

get_content_type_charset(*default='UTF-8'*)
Inspect the Content-Type header to retrieve the charset that the client has specified.

Parameters **default** (*str*) – The default charset to return if none exists.

Returns The charset of the request.

Return type str

get_query(*name*, *default=None*)
Get a value from the query data that was sent to the server.

Parameters

- **name** (*str*) – The name of the query value to retrieve.
- **default** – The value to return if *name* is not specified.

Returns The value if it exists, otherwise *default* will be returned.

Return type str

guess_mime_type(*path*)
Guess an appropriate MIME type based on the extension of the provided path.

Parameters **path** (*str*) – The of the file to analyze.

Returns The guessed MIME type of the default if non are found.

Return type str

handler_map = None
The dictionary object which maps regular expressions of resources to the functions which should handle them.

headers_active = None
Whether or not the request is in the sending headers phase.

log_error(*msg_format*, **args*)
Log an error.

This is called when a request cannot be fulfilled. By default it passes the message on to log_message().

Arguments are the same as for log_message().

XXX This should go to the separate error log.

log_message(*msg_format*, **args*)
Log an arbitrary message.

This is used by all other logging functions. Override it if you have specific logging wishes.

The first argument, FORMAT, is a format string for the message to be logged. If the format string contains any % escapes requiring parameters, they should be specified as subsequent arguments (it's just like printf!).

The client ip address and current date/time are prefixed to every message.

on_init()
This method is meant to be over ridden by custom classes. It is called as part of the __init__ method and provides an opportunity for the handler maps to be populated with entries or the config to be customized.

query_data = None
The parameter data that has been passed to the server parsed as a dict.

raw_query_data = None
The raw data that was parsed into the `query_data` attribute.

respond_file(*file_path*, *attachment=False*, *query=None*)
Respond to the client by serving a file, either directly or as an attachment.

Parameters

- **file_path** (`str`) – The path to the file to serve, this does not need to be in the web root.
- **attachment** (`bool`) – Whether to serve the file as a download by setting the Content-Disposition header.

respond_list_directory(*dir_path*, *query=None*)
Respond to the client with an HTML page listing the contents of the specified directory.

Parameters **dir_path** (`str`) – The path of the directory to list the contents of.

respond_not_found()
Respond to the client with a default 404 message.

respond_redirect(*location='/'*)
Respond to the client with a 301 message and redirect them with a Location header.

Parameters **location** (`str`) – The new location to redirect the client to.

respond_server_error(*status=None*, *status_line=None*, *message=None*)
Handle an internal server error, logging a traceback if executed within an exception handler.

Parameters

- **status** (`int`) – The status code to respond to the client with.
- **status_line** (`str`) – The status message to respond to the client with.
- **message** (`str`) – The body of the response that is sent to the client.

respond_unauthorized(*request_authentication=False*)
Respond to the client that the request is unauthorized.

Parameters **request_authentication** (`bool`) – Whether to request basic authentication information by sending a WWW-Authenticate header.

rpc_handler_map = None
The dictionary object which maps regular expressions of RPC functions to their handlers.

send_response(**args*, ***kwargs*)
Send the response header and log the response code.

Also send two standard headers with the server software version and the current date.

stock_handler_respond_not_found(*query*)
This method provides a handler suitable to be used in the handler_map.

stock_handler_respond_unauthorized(*query*)
This method provides a handler suitable to be used in the handler_map.

version_string()
Return the server software version string.

web_socket_handler = None
An optional class to handle Web Sockets. This class must be derived from `WebSocketHandler`.

class RPCClient(*address*, *use_ssl=False*, *username=None*, *password=None*, *uri_base='/'*, *ssl_context=None*)

This object facilitates communication with remote RPC methods as provided by a `RequestHandler` instance. Once created this object can be called directly, doing so is the same as using the call method.

This object uses locks internally to be thread safe. Only one thread can execute a function at a time.

__init__(*address*, *use_ssl=False*, *username=None*, *password=None*, *uri_base='/'*, *ssl_context=None*)

Parameters

- **address** (`tuple`) – The address of the server to connect to as (host, port).
- **use_ssl** (`bool`) – Whether to connect with SSL or not.
- **username** (`str`) – The username to authenticate with.
- **password** (`str`) – The password to authenticate with.
- **uri_base** (`str`) – An optional prefix for all methods.
- **ssl_context** – An optional SSL context to use for SSL related options.

call(*method*, **args*, ***kwargs*)

Issue a call to the remote end point to execute the specified procedure.

Parameters **method** (`str`) – The name of the remote procedure to execute.

Returns The return value from the remote function.

decode(*data*)

Decode data with the configured serializer.

encode(*data*)

Encode data with the configured serializer.

headers = None

An optional dictionary of headers to include with each RPC request.

lock = None

A `threading.Lock` instance used to synchronize operations.

reconnect()

Reconnect to the remote server.

serializer = None

The `Serializer` instance to use for encoding RPC data to the server.

set_serializer(*serializer_name*, *compression=None*)

Configure the serializer to use for communication with the server. The serializer specified must be valid and in the `g_serializer_drivers` map.

Parameters

- **serializer_name** (`str`) – The name of the serializer to use.
- **compression** (`str`) – The name of a compression library to use.

class RPCClientCached(**args*, ***kwargs*)

This object builds upon `RPCClient` and provides additional methods for cacheing results in memory.

cache_call(*method*, **options*)

Call a remote method and store the result locally. Subsequent calls to the same method with the same arguments will return the cached result without invoking the remote procedure. Cached results are kept indefinitely and must be manually refreshed with a call to `cache_call_refresh()`.

Parameters **`method`** (*`str`*) – The name of the remote procedure to execute.

Returns The return value from the remote function.

`cache_call_refresh`(*method*, **options*)

Call a remote method and update the local cache with the result if it already existed.

Parameters **`method`** (*`str`*) – The name of the remote procedure to execute.

Returns The return value from the remote function.

`cache_clear`()

Purge the local store of all cached function information.

`class Serializer`(*name*, *charset='UTF-8'*, *compression=None*)

This class represents a serilizer object for use with the RPC system.

`__init__`(*name*, *charset='UTF-8'*, *compression=None*)

Parameters

- **`name`** (*`str`*) – The name of the serializer to use.
- **`charset`** (*`str`*) – The name of the encoding to use.
- **`compression`** (*`str`*) – The compression library to use.

`dumps`(*data*)

Serialize a python data type for transmission or storage.

Parameters **`data`** – The python object to serialize.

Returns The serialized representation of the object.

Return type bytes

`classmethod from_content_type`(*content_type*)

Build a serializer object from a MIME Content-Type string.

Parameters **`content_type`** (*`str`*) – The Content-Type string to parse.

Returns A new serializer instance.

Return type *`Serializer`*

`loads`(*data*)

Deserialize the data into it's original python object.

Parameters **`data`** (*`bytes`*) – The serialized object to load.

Returns The original python object.

`class ServerNonThreaded`(**args*, ***kwargs*)

This class is used internally by *`AdvancedHTTPServer`* and is not intended for use by other classes or functions. It is responsible for listening on a single address, TCP port and SSL combination.

`finish_request`(*request*, *client_address*)

Finish one request by instantiating RequestHandlerClass.

`get_request`()

Get the request and client address from the socket.

May be overridden.

`handle_request`()

Handle one request, possibly blocking.

Respects self.timeout.

server_bind(**args*, ***kwargs*)
Override server_bind to store the server name.

shutdown(**args*, ***kwargs*)
Stops the serve_forever loop.

Blocks until the loop has finished. This must be called while serve_forever() is running in another thread, or it will deadlock.

class ServerTestCase(**args*, ***kwargs*)
A base class for unit tests with AdvancedHTTPServer derived classes.

assertHTTPStatus(*http_response*, *status*)
Check an HTTP response object and ensure the status is correct.

Parameters

- **http_response** (`http.client.HTTPResponse`) – The response object to check.
- **status** (*int*) – The status code to expect for *http_response*.

handler_class
The *RequestHandler* class to use as the request handler, this can be overridden by subclasses.

alias of *RequestHandler*

http_request(*resource*, *method='GET'*, *headers=None*)
Make an HTTP request to the test server and return the response.

Parameters

- **resource** (*str*) – The resource to issue the request to.
- **method** (*str*) – The HTTP verb to use (GET, HEAD, POST etc.).
- **headers** (*dict*) – The HTTP headers to provide in the request.

Returns The HTTP response object.

Return type `http.client.HTTPResponse`

server_class
The *AdvancedHTTPServer* class to use as the server, this can be overridden by subclasses.

alias of *AdvancedHTTPServer*

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_resource = None
A resource which has a handler set to it which will respond with a 200 status code and the message 'Hello World!'

class ServerThreaded(**args*, ***kwargs*)
This class is used internally by *AdvancedHTTPServer* and is not intended for use by other classes or functions. It is responsible for listening on a single address, TCP port and SSL combination.

class SSLSNICertificate(*hostname*, *certfile*, *keyfile*)
The information for a certificate used by SSL's Server Name Indicator (SNI) extension.

New in version 2.2.0.

hostname
The hostname string for requests which should use this certificate information.

certfile
The path to the SSL certificate file on disk to use for the hostname.

keyfile
The path to the SSL key file on disk to use for the hostname.

class WebSocketHandler(*handler*)
A handler for web socket connections.

close()
Close the web socket connection and stop processing results. If the connection is still open, a WebSocket close message will be sent to the peer.

on_closed()
A method that can be over ridden and is called after the web socket is closed.

on_connected()
A method that can be over ridden and is called after the web socket is connected.

on_message(*opcode*, *message*)
The primary dispatch function to handle incoming WebSocket messages.

Parameters

- **opcode** (`int`) – The opcode of the message that was received.
- **message** (`bytes`) – The data contained within the message.

on_message_binary(*message*)
A method that can be over ridden and is called when a binary message is received from the peer.

Parameters **message** (`bytes`) – The message data.

on_message_text(*message*)
A method that can be over ridden and is called when a text message is received from the peer.

Parameters **message** (`str`) – The message data.

send_message(*opcode*, *message*)
Send a message to the peer over the socket.

Parameters

- **opcode** (`int`) – The opcode for the message to send.
- **message** (`bytes`) – The message data to send.

1.2.4 Exceptions

exception RPCError(*message*, *status=None*, *remote_exception=None*)
This class represents an RPC error either local or remote. Any errors in routines executed on the server will raise this error.

is_remote_exception
This is true if the represented error resulted from an exception on the remote server.

Type bool

exception RPCConnectionError(*message*, *status=None*, *remote_exception=None*)
An exception raised when there is a connection-related error encountered by the RPC client.

New in version 2.1.0.

Python Module Index

Index