Mod_python Servlets

API Documentation

March 15, 2005

Contents

Contents **1**

1 Module servlet **2**

1.1 Functions 2
1.2 Class HTMLPage 3
1.2.1 Methods 3
1.2.2 Class Methods 5
1.2.3 Instance Variables 5
1.3 Class Servlet 7
1.3.1 Methods 7
1.3.2 Class Methods 10
1.3.3 Instance Variables 10

Index **14**

1 Module servlet

Copyright 2004 Daniel J. Popowich

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

`http://www.apache.org/licenses/LICENSE-2.0`

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

See `handler` for details about the servlet handler.

$Id: servlet.py,v 1.12 2005/02/12 04:10:00 popowich Exp $

1.1 Functions

handler(*req*)

This handler uses instances of subclasses of `Servlet` to handle HTTP GET and POST requests. These instances are referred to as *servlets* in this documentation.

Overview of how it works:

1. A developer creates a class that subclasses `Servlet` or, more probably, `HTMLPage`, which already subclasses `Servlet` and has many features for generating HTML.
2. Let's say we have a class named `Foo` that subclasses `HTMLPage`. This class must be saved in a file called Foo.mps. (mps stands for Mod Python Servlet).

   Generally, for any servlet named SERVLET, it must be a subclass of `Servlet` and saved in a file named SERVLET.mps.
3. Apache needs to be configured to call this handler for Foo.mps. Assuming /var/www/html is your DocumentRoot and also assuming Foo.mps is saved in directory /var/www/html/mps_test then Apache needs the following configuration:

   ```
   <Directory /var/www/html/mps_test>
     SetHandler mod_python
     PythonHandler mod_python.servlet
     PythonDebug on
   </Directory>
   ```
4. When the server receives a request for */mps_test/Foo* it will look for a file named Foo.mps in /var/www/html/mps_test, compile the file, look for a class named Foo, insure it is a subclass of `Servlet`, create an instance of this class, let's call it `servlet`, and then call, in order:
   (a) `servlet.auth()`
   (b) `servlet.prep()`
   (c) `servlet.respond()`
   (d) `servlet.wrapup()`

   See the documentation for the above methods of `Servlet` for details about each stage of processing the request.
5. Instances of a class are cached, so if a request for */mps_test/Foo* comes in another request, the instance will be found in the cache and reused.

See the documentation for `Servlet` and `HTMLPage` for further details about servlet processing and helpful features.

If installed, see the **`tutorial`** for a live demonstration of servlets. See the README file that came with this distribution for tutorial installation instructions.

1.2 Class HTMLPage

servlet.Servlet
HTMLPage

`HTMLPage` is the servlet that most developers wanting to generate HTML will want to subclass. The `respond` method of `HTMLPage` calls `write_html` which writes to the client a well-defined HTML document.

Many methods and instance variables offer features to the developer to allow the customization of their content. Use of subclassing can produce site-wide continuity of content; e.g., you can create a servlet called `SitePage` (that inherits from `HTMLPage`) that will provide your site-wide look and feel, then individual servlets can inherit from `SitePage`.

To see how `HTMLPage` writes out an HTML document, look at `write_html` and the methods it calls.

1.2.1 Methods

respond(*self*)

This first calls `Servlet.repsond` (to see if a method should be called; see `ok_methods_to_call`) and if it returns False, calls `write_html`.

Overrides: servlet.Servlet.respond

write_base(*self*)

Write BASE tag to the client in the HEAD.

See Also: `base`

write_body(*self*)

Writes the BODY section to the client.
This method writes "<BODY...>" to the client taking into account the `body_attrs` instance variable. It then calls `write_body_parts` and finally writes the "</BODY>" tag.
This method will *not* be typically overridden. To modify the content of the BODY section, see `write_body_parts`.

write_body_parts(*self*)

Write the content of the BODY section.
The base implementation simply calls `write_content`.
For complex pages that have many sections, this method will typically be overridden, e.g., write to the client the layout of the site look and feel and where page-specific content should appear, call `write_content`.
See the **tutorial** that comes with the distribution for examples of how this method can be used to create site-wide content.

write_content(*self*)

This method must be overridden to produce content on the page.

write_css(*self*)

Writes CSS to the client in the HEAD.

See Also: `css_links`, `css`

write_doctype(*self*)

Writes to the client the <!DOCTYPE...> tag. By default, the following is used:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
```

This can be controlled by setting the `doctype` instance variable. If set, it will be written, as-is, to the client. If it is not set, then the `dtd` instance variable will be checked; it can be one of ”strict”, ”loose” or ”frameset”. The default is ”loose” and the resulting DOCTYPE is shown above. If set to ”strict” or ”frameset” an appropriate DOCTYPE will be generated.
This method will *not* be typically overridden.

write_head(*self*)

This writes the HEAD section of the HTML document to the client.
It first writes ”<HEAD”> then calls `write_head_parts` and, lastly, writes ”</HEAD>”.
This method will *not* be typically overridden. To modify the content of the HEAD section, see `write_head_parts`.

write_head_parts(*self*)

This method calls, in order:

1. `write_title`
2. `write_shortcut_icon`
3. `write_base`
4. `write_meta`
5. `write_css`
6. `write_js`

See the above methods for details about what each writes to the client. All of the above methods can have the content they write to the client controlled by setting (or unsetting) instance variables.
While the base method will serve most developer needs, this method is a likely candidate to be overridden. If you want to *add* to the HEAD you will likely want to call the superclass method and then write additional content in your method, e.g.:

```
class MyServlet(HTMLPage):
    ...

    def write_head_parts(self):
        HTMLPage.write_head_parts(self)
        # add my own content
        self.writeln(...)
```

write_html(*self*)

This method produces a well-formed HTML document and writes it to the client.
It first calls `write_doctype` which writes the DOCTYPE. It then writes "<HTML>". In turn it calls `write_head` and `write_body` which write the HEAD and BODY sections, respectively. It finishes by writing "</HTML>".

Return Value
True

write_js(*self*)

Writes javascript to the client in the HEAD.

See Also: `js_src`, `js`

write_meta(*self*)

Writes META tags to the client in the HEAD.

See Also: `meta`, `http_equiv`

write_shortcut_icon(*self*)

Writes a LINK tag to the client specifying the shortcut icon.

See Also: `shortcut_icon`

write_title(*self*)

Writes the TITLE tag to the client.
It checks for the `title` instance variable. If it does not exist it uses the name of the servlet for the title. If `title` is callable it will be called (with no arguments) and its output will be used as the title.

See Also: `title`

Inherited from Servlet: __init__, __str__, add_cookie, auth, external_redirect, flush, get_cookies, internal_redirect, log, prep, raw_write, wrapup, write, writeln

1.2.2 Class Methods

Inherited from Servlet: count, sourcefilename

1.2.3 Instance Variables

Name	Description
base	**Value:** `''` If set, this will specify the BASE tag in the HEAD. If `base` is a string it will be the value of the `href` attribute. If `base` is a tuple, it should have two string elements of the form (*href*, *target*) which will be the values of those BASE tag attributes.
body_attrs	**Value:** {} Attributes for BODY tag. Default is empty dict. If non empty, the keys will become attribute names for the BODY tag and the values will be the attribute values.

continued on next page

Name	Description
content_type	**Value:** `'text/html'` Specifies the content type of the servlet, i.e., ”text/html”. If it is not set (None) then it defaults to ”text/plain”. Default: None
css	**Value:** `''` A string or list of strings of CSS. If non empty, a <STYLE type=”text/css”>...</STYLE> section will be placed in the HEAD. If css is a list of strings, the elements will be joined (seperated by newlines) and placed in a single <STYLE ...>...</STYLE> section.
css_links	**Value:** `[]` A list of hrefs (strings). For each href, a <LINK type=”text/css” rel=”stylesheet” ...> will be placed in the HEAD.
doctype	See `write_doctype`
dtd	See `write_doctype`
http_equiv	**Value:** {} http_equiv, like `meta`, produces META tags in the HEAD, but instead of the keys being the value of the `name` attribute, they are the value of the `http-equiv` attribute. See `meta` for details.
js	**Value:** `''` A string or list of strings of javascript. If non empty, a <SCRIPT type=”text/javascript”>...</SCRIPT> section will be placed in the HEAD. If js is a list of strings, the elements will be joined (seperated by newlines) and placed in a single <SCRIPT ...>...</SCRIPT> section.
js_src	**Value:** `[]` A list of hrefs (strings). For each href, a <SCRIPT type=”text/javascript” src=”HREF”></SCRIPT> will be placed in the HEAD.

meta

Value: {}

A dict, which if non-empty will produce META tags in the HEAD. The keys of the dict will be the values of the `name` attribute and their values will become the `content` attribute. For example, the following value of meta:

```
meta = {"Author" : "Daniel Popowich",
        "Date"   : "April 28, 2004"}
```

will produce the following output in the HEAD:

```
<META name="Author" content="Daniel Popowich">
<META name="Date" content="April 28, 2004">
```

The values of the dict may be a list or tuple of strings, in which case multiple META tags will be produced for the same `name` attribute. For example:

```
meta = {"Author" : ["Tom", "Jerry"]}
```

will produce the following output in the HEAD:

```
<META name="Author" content="Tom">
<META name="Author" content="Jerry">
```

Name	Description
shortcut_icon	**Value:** `''` If non empty, specifies the href to a shortcut icon and produces a LINK tag in the HEAD: <LINK rel=”shortcut icon” href=”*shortcut_icon*”>

continued on next page

Name	Description
title	The title of the HTML page. If not set, it defaults to the name of the servlet. `title` can be a callable that accepts no arguments, in which case the title will be the output of the callable.
Inherited from Servlet: auth_realm *(p. 7)*, form *(p. 7)*, form_vars *(p. 7)*, instantiated *(p. 7)*, ok_methods_to_call *(p. 7)*, path_info *(p. 7)*, query_vars *(p. 7)*, req *(p. 7)*, reusable *(p. 7)*, session *(p. 7)*, session_timeout *(p. 7)*, use_session *(p. 7)*	

1.3 Class Servlet

Known Subclasses: HTMLPage

Abstract base class for all mod_python servlets.

You must subclass this class implementing, minimally, `respond`, but you will probably want to subclass `HTMLPage`, which already subclasses Servlet and has many added features for generating HTML output.

1.3.1 Methods

__init__(*self*)

Base constructor for all servlets.
If subclasses override this method they *must* call this method for proper servlet initialization. For example:

```
class MyServlet(Servlet):

    def __init__(self):
        Servlet.__init__(self)
        ...
```

__str__(*self*)

Return a simple string representing an instance:
"**Servlet *class name***"

add_cookie(*self*, *cookie*, *value*=`''`, ***kw*)

Wrapper around mod_python.Cookie.add_cookie(). See mod_python documentation for details.

auth(*self*)

Basic HTTP authentication method.
This method is the first user method called by the `handler` for each request, just before `prep`. It should return without exception if authorization is granted (the return value is ignored) or raise `apache.SERVER_RETURN` with `apache.HTTP_UNAUTHORIZED` as a value if authorization is denied. Typical implementation should use the `_get_user_pw` and `_unauthorized` helper methods:

```
def auth(self):
    user, pw = self._get_user_pw()
    # test user, pw for authorization; if OK, return
    self._unauthorized()
```

See the source code for `_auth_with_mapping` for a simple implementation.
The base method is a no-op, i.e., no authentication is required.

external_redirect(*self*, *uri*, *permanently*=`True`)

Send a redirect to the client via a **Location** HTTP header.

Parameters

`uri:`	The URI to relocate to. *(type=str)*
`permanently:`	If True (the default) reply with 301 (MOVED_PERMANENTLY, else 302 (MOVED_TEMPORARILY). *(type=bool)*

Return Value

Does not return.

flush(*self*)

Utilily method which immediately flushes all buffered output to the client.
If `content_type` has not already been written it will be.

Return Value

None

get_cookies(*self*, *klass*=<`class 'mod_python.Cookie.Cookie'`>, ***kw*)

Wrapper around mod_python.Cookie.get_cookies(). See mod_python documentaion for details.

internal_redirect(*self*, *uri*)

Redirect internally. This does not send a redirect to the client (see `external_redirect`), but redirects internally to the server. This is a wrapper around req.internal_redirect(); see mod_python documentation for details.

Parameters

`uri:`	The URI to relocate to. *(type=str)*

Return Value

Does not return.

log(*self*, *msg*)

Utility method to write a message to the apache error log.

Parameters

`msg:`	The message to be written to the log. *(type=str or an object that can be converted into a str.)*

Return Value

None

prep(*self*)

This is the second user method called by the `handler`, after `auth`, prior to `respond`.
It should be used as a means to prep the servlet for `respond`, e.g., opening data files, acquiring db connections, preprocessing form data, etc.
The return value is ignored by the handler.
If this method is implemented in a subclass, the superclass method *must* be called; e.g:

```
class MyServlet(HTMLPage):
    ...

    def prep(self):
        HTMLPage.prep(self)
        ...
    ...
```

raw_write(*self*, **args*)

Like `write`, but all output is immediately flushed to the client and no string coercion is attempted on the arguments.

Parameters

`args:` tuple of arbitrary objects

Return Value

None

respond(*self*)

This is the third user method called by the `handler`, after `prep`, before `wrapup` and is where the response is generated. Typically, this will be by calls to `write`, `writeln` and/or `raw_write`. For example:

```
def respond(self):
    self.writeln("Hello, world!")
```

For most developers, this method will not need to be written because the version implemented in `HTMLPage` is sufficient.
The base class implementation (which is called by `HTMLPage.respond`) allows for arbitrary methods to be called during form POSTs. See `ok_methods_to_call`.
This method must return True or False. If True, request processing will continue (and ultimately, content sent to the client), if False, a 204 response will be sent to the client (NO_CONTENT) and processing will stop.

Return Value

bool

wrapup(*self*)

This is the fourth user method called by the `handler`, immediately after calling `respond`, before `_finally`.
This method should be used to ”tidy up” after a request, e.g., flush output data, create a log entry, etc.
The base method flushes output to the client. If overridden, this method should be called by the subclass if it does not call `flush`.
The return value is ignored by the handler.

write(*self*, **args*)

Utility method to write the arguments to the client.
Each arg is coerced to a string and buffered for output. The output is sent to the client when `flush` is called explicitly or implicitly, when the handler is finished with the request.

Parameters
`args:` tuple of arbitrary objects

Return Value
None

writeln(*self*, **args*)

Like `write`, but in addition appends a newline to the output.

Parameters
`args:` tuple of arbitrary objects

Return Value
None

1.3.2 Class Methods

count(*klass*)

Get the count of requests for this servlet.

Return Value
int

sourcefilename(*klass*)

Get the filename of the source file for this servlet.

Return Value
str

1.3.3 Instance Variables

Name	Description
auth_realm	**Value:** `'Unspecified'` Specifies the realm for basic HTTP authentication. Default: "Unspecified".
content_type	**Value:** `None` Specifies the content type of the servlet, i.e., "text/html". If it is not set (None) then it defaults to "text/plain". Default: None
form	User data sent with request via form or url. For each request an instance of FieldStorage is created. Note: FieldStorage is created with blank values being ignored, ie, as if they did not exist.
form_vars	**Value:** `[]` This is the same as `query_vars` except these variables are only processed for POST requests. For *all* GET requests, these variables will be set to their default values.

continued on next page

Name	Description
instantiated	Timestamp of when the servlet was instantiated. This is stored as seconds since the epoch; see the python documentation for time.time().

ok_methods_to_call	**Value:** `[]`
	list of methods that can be called directly via a POST. If the name of a form element looks like:
	`_call_NAME()`
	and NAME is the name of an unbound method of the servlet and the method is listed in ok_methods_to_call, then that method will be called when the form is posted. The form element can also look like this:
	`_call_SOME_METHOD(1,2,3)`
	and SOME_METHOD will be called with three arguments: 1, 2, 3. See the **tutorial** that comes with the distribution for examples.

path_info	req.path_info canonicalized as a list: stripping beginning and trailing "/" and splitting it on internal "/".

continued on next page

Name	Description

query_vars	**Value:** `[]`
	List of arguments to be searched for in `form` to be set as instance variables of the servlet. This processing occurs during the call to `prep`.
	query_vars can be either a list of strings (the variable names) or a list of tuples of the form:
	`(NAME, DEFAULT [,CONVERSION])`
	where NAME, a string, is the name of the variable and must be a legal python identifier; DEFAULT, must be a string, list or dict, is the default value for NAME if it does not appear in `form`, and, optionally, CONVERSION is a callable that can convert the string found in `form` (or DEFAULT) to an appropriate value and/or type.
	When an element of query_var is just a string (not a tuple), this is equivalent to:
	`(NAME, '')`
	That is, the default value for NAME, if not found in `form`, is an empty string.
	If DEFAULT is a string, the value will be retrieved from `form` with a call to form.getfirst(). If DEFAULT is a list, the value will be retrieved by a call to form.getlist(). If DEFAULT is a dict, form will be searched for names with the pattern NAME[KEY] with calls to either getfirst() or getlist() depending on whether the default dict value is a list or string for that key.
	All string values retrieved from `form` will be stripped of whitespace.
	Example of query_vars:

```
["name",
 ("login", "", bool),
 ("items", []),
 ("map", {"foo" : "bar",
          "baz" : []})]
```

Name	Description
	query_vars should be contrasted to `form_vars`. Where `form_vars` are only processed for POST requests, query_vars is processed for *both* POST and GET requests. If installed, see the **tutorial** for a live demonstration of query_vars.
req	The apache request object.
reusable	**Value:** `True` Flag (default: True) indicating whether or not an instance of this servlet can be used for multiple requests. If False, instances will be recreated for every request of a servlet.
session	**Value:** `None` A Session (see the mod_python documentation for mod_python.Session.Session). A session object is created for each request if `use_session` is True. If `use_session` is False, this variable will be set to None.
session_timeout	**Value:** `1800` Length of time, in seconds, until session times out. Default: 30 minutes.
use_session	**Value:** `False` If true, create (or reload) `session` for each request. Default: False.

continued on next page

Name	Description

Index

servlet *(module)*, 2–13
  handler *(function)*, 2
  HTMLPage *(class)*, 2–7
    respond *(method)*, 3
    write_base *(method)*, 3
    write_body *(method)*, 3
    write_body_parts *(method)*, 3
    write_content *(method)*, 3
    write_css *(method)*, 3
    write_doctype *(method)*, 4
    write_head *(method)*, 4
    write_head_parts *(method)*, 4
    write_html *(method)*, 4
    write_js *(method)*, 5
    write_meta *(method)*, 5
    write_shortcut_icon *(method)*, 5
    write_title *(method)*, 5
  Servlet *(class)*, 7–13
    __init__ *(method)*, 7
    __str__ *(method)*, 7
    add_cookie *(method)*, 7
    auth *(method)*, 7
    count *(method)*, 10
    external_redirect *(method)*, 7
    flush *(method)*, 8
    get_cookies *(method)*, 8
    internal_redirect *(method)*, 8
    log *(method)*, 8
    prep *(method)*, 8
    raw_write *(method)*, 9
    respond *(method)*, 9
    sourcefilename *(method)*, 10
    wrapup *(method)*, 9
    write *(method)*, 9
    writeln *(method)*, 10