

Bottle Documentation

This document is a **work in progress** and intended to be a tutorial, howto and an api documentation at the same time. If you have questions not answered here, please check the [F.A.Q.](#) or file a ticket at bottles [issue tracker](#).

This documentation describes the features of the **0.6.4 Release** and is not updated anymore. The 0.7 docs can be found [here](#)

"Hello World" in a Bottle

Lets start with a very basic example: Hello World

```
1 from bottle import route, run
2 @route('/hello')
3 def hello():
4     return "Hello World!"
5 run() # This starts the HTTP server
```

Run this script, visit <http://localhost:8080/hello> and you will see "Hello World!" in your Browser. So, what happened here?

1. First we imported some bottle components. The `route()` decorator and the `run()` function.
2. The `route()` [decorator](#) is used do bind a piece of code to an URL. In this example we want to answer requests to the `/hello` URL.
3. This function will be called every time someone hits the `/hello` URL on the web server. It is called a **handler function** or **callback**.
4. The return value of a handler function will be sent back to the Browser.
5. Now it is time to start the actual HTTP server. The default is a development server running on `localhost` port `8080` and serving requests until you hit **Ctrl-C**

Routing

Routes are used to map URLs to **callbacks** that generate the content for the specific URL. Bottle has a `route()` decorator to do that. You can add any number of routes to a callback.

```
1 from bottle import route
2 @route('/')
3 @route('/index.html')
4 def index():
5     return "<a href='/hello'>Go to Hello World page</a>"
6
7 @route('/hello')
8 def hello():
9     return "Hello World!"
```

As you can see, URLs and routes have nothing to do with actual files on the web server. Routes are unique names for your callbacks, nothing more and nothing less. Requests to URLs not matching any routes are answered with a 404 HTTP error. Exceptions within your handler callbacks will cause a 500 error.

Request Methods

The `route()` decorator has an optional keyword argument `method` which defaults to `method='GET'`, so only GET requests get answered. Possible values are POST, PUT, DELETE, HEAD or any other [HTTP request method](#) you want to listen to.

```
1 from bottle import route, request
2 @route('/form/submit', method='POST')
3 def form_submit():
4     form_data = request.POST
5     do_something_with(form_data)
6     return "Done"
```

In this example we used `request.POST` to access POST form data. This is described [here](#)

Dynamic Routes

Static routes are fine, but URLs may carry information as well. Let's add a `:name` placeholder to our route.

```
1 from bottle import route
2 @route('/hello/:name')
3 def hello(name):
4     return "Hello %s!" % name
```

This dynamic route matches `/hello/alice` as well as `/hello/bob`. In fact, the `:name` part of the route matches everything but a slash (`/`), so any name is possible. `/hello/bob/and/alice` or `/hellobob` won't match.

Each part of the URL covered by a placeholder is provided as a keyword parameter to your handler callback.

Regular Expressions

The default placeholder matches everything up to the next slash. To change that, you can add some regular expression:

```
1 from bottle import route
2 @route('/get_object/:id#[0-9]+#')
3 def get(id):
4     return "Object ID: %d" % int(id)
```

or even use full featured regular expressions with named groups:

```
1 from bottle import route
2 @route('/get_object/(?P<id>[0-9]+)')
3 def get(id):
4     return "Object ID: %d" % int(id)
```

As you can see, URL parameters remain strings, even if they are configured to only match digits. You have to explicitly cast them into the type you need.

The `@validate()` decorator

Bottle offers a handy decorator called `validate()` to check and manipulate URL parameters. It takes callables (function or class objects) as keyword arguments and filters every URL parameter through the corresponding callable before they are passed to your request handler.

```

1 from bottle import route, validate
2 # /test/validate/1/2.3/4,5,6,7
3 @route('/test/validate/:i/:f/:csv')
4 @validate(i=int, f=float, csv=lambda x: map(int, x.split(',')))
5 def validate_test(i, f, csv):
6     return "Int: %d, Float:%f, List:%s" % (i, f, repr(csv))

```

You may raise `ValueError` in your custom callable if a parameter does not validate.

Generating content

TODO

Output Casting

The [WSGI specification](#) expects an iterable list of byte strings to be returned from your application and can't handle file objects, unicode, dictionaries or exceptions.

```

1 from bottle import route
2 @route('/wsgi')
3 def wsgi():
4     return ['WSGI', 'wants a', 'list of', 'strings']

```

Bottle automatically tries to convert anything to a WSGI supported type, so you don't have to. The following examples will work with Bottle, but won't work with pure WSGI.

Strings and Unicode

Returning strings (bytes) is not a problem. Unicode however needs to be encoded into a byte stream before the webserver can send it to the client. The default encoding is utf-8, so if that fits your needs, you can simply return unicode or unicode iterables.

```

1 from bottle import route, response
2 @route('/string')
3 def get_string():
4     return 'Bottle converts strings to iterables'
5
6 @route('/unicode')
7 def get_unicode():
8     return u'Unicode is encoded with UTF-8 by default'

```

You can change Bottle's default encoding by setting `response.content_type` to a value containing a `charset=...` parameter or by changing `response.charset` directly.

```

1 from bottle import route, response
2 @route('/iso')
3 def get_iso():
4     response.charset = 'ISO-8859-15'
5     return u'This will be sent with ISO-8859-15 encoding.'
6
7 @route('/latin9')
8 def get_latin():
9     response.content_type = 'text/html; charset=latin9'
10    return u'ISO-8859-15 is also known as latin9.'

```

In some rare cases the Python encoding names differ from the names supported by the HTTP

specification. Then, you have to do both: First set the `response.content_type` header (which is sent to the client unchanged) and then set the `response.charset` option (which is used to decode unicode).

File Objects and Streams

Bottle wraps everything that has a `read()` method (file objects) with the `wsgi.file_wrapper` provided by your WSGI server implementation. This wrapper should use highly optimised system calls for your operating system (`sendfile` on UNIX) to transfer the file.

```
1 @route('/file')
2 def get_file():
3     return open('some/file.txt', 'r')
```

JSON

Even dictionaries are allowed. They are converted to [json](#) and returned with `Content-Type` header set to `application/json`. To disable this feature (and pass dicts to your middleware) you can set `bottle.default_app().autojson` to `False`.

```
1 @route('/api/status')
2 def api_status():
3     return {'status': 'online', 'servertime': time.time() }
```

Static Files

You can directly return file objects, but `bottle.send_file()` is the recommended way to serve static files. It automatically guesses a mime-type, adds a `Last-Modified` header, restricts paths to a `root` directory for security reasons and generates appropriate error pages (401 on permission errors, 404 on missing files). It even supports the `If-Modified-Since` header and eventually generates a `304 Not modified` response. You can pass a custom mimetype to disable mimetype guessing.

```
1 from bottle import send_file
2
3 @route('/static/:filename')
4 def static_file(filename):
5     send_file(filename, root='/path/to/static/files')
6
7 @route('/images/:filename#.*\.png#')
8 def static_image(filename):
9     send_file(filename, root='/path/to/image/files', mimetype='image/png')
```

HTTP Errors and Redirects

The `bottle.abort(code[, message])` function is used to generate [HTTP error pages](#).

```
1 from bottle import route, redirect, abort
2 @route('/restricted')
3 def restricted():
4     abort(401, "Sorry, access denied.")
```

To redirect a client to a different URL, you can send a `307 Temporary Redirect` response with the `Location` header set to the new URL. `bottle.redirect(url[, code])` does that for you. You may provide a different HTTP status code as a second parameter.

```
1 from bottle import route, redirect, abort
2 @route('/wrong/url')
3 def wrong():
4     redirect("/right/url")
```

Both functions interrupt your handler code (by throwing a `bottle.HTTPError` exception) so you don't have to return anything.

All unhandled exceptions other than `bottle.HTTPError` will result in a `500 Internal Server Error` response, so they won't crash your WSGI server.

HTTP Stuff

TODO

Cookies

Bottle stores cookies sent by the client in a dictionary called `request.COOKIES`. To create new cookies, the method `response.set_cookie(name, value[, **params])` is used. It accepts additional parameters as long as they are valid cookie attributes supported by [SimpleCookie](#).

```
1 from bottle import response
2 response.set_cookie('key', 'value', path='/', domain='example.com', secure=True, expires=)
```

To set the `max-age` attribute use the `max_age` name.

GET and POST values

Query strings and/or POST form submissions are parsed into dictionaries and made available as `bottle.request.GET` and `bottle.request.POST`. Multiple values per key are possible, so each each value of these dictionaries may contain a string or a list of strings.

```
1 <form action="/search" method="post">
2   <input type="text" name="query" />
3   <input type="submit" />
4 </form>
5
6 #!Python
7 from bottle import route, request
8 @route('/search', method='POST')
9 def do_search():
10     query = request.POST.get('query', '').strip()
11     if not query:
12         return "You didn't supply a search query."
13     else:
14         return 'You searched for %s.' % query
```

File Uploads

```
1 Bottle handles file uploads similar to normal POST form data.
2 Instead of strings or list of strings, you will get file-like objects.
3
4 #!html
5 <form action="/upload" method="post" enctype="multipart/form-data">
6   <input name="datafile" type="file" />
7 </form>
8
9 #!Python
10 from bottle import route, request
11 @route('/upload', method='POST')
12 def do_upload():
13     datafile = request.POST.get('datafile')
14     return datafile.read()
```

Templates

Bottle uses its own little template engine by default. You can use a template by calling `template(template_name, **template_arguments)` and returning the result.

```
1 @route('/hello/:name')
2 def hello(name):
3     return template('hello_template', username=name)
```

The `@view` decorator is another option:

```
1 @route('/hello/:name')
2 @view('hello_template')
3 def hello(name):
4     return dict(username=name)
```

Both examples load the template `hello_template.tpl` with the `username` variable set to the URL `:name` part and return the result as a string.

A simple `hello_template.tpl` file looks this:

```
1 <h1>Hello {{username}}</h1>
2 <p>How are you?</p>
```

Template search path

The list `bottle.TEMPLATE_PATH` is used to map template names to actual file names. By default, this list contains `['./', './views/']`.

Template caching

Templates are cached in memory after compilation. Modifications made to the template file will have no affect until you clear the template cache. Call `bottle.TEMPLATES.clear()` to do so.

Template Syntax

An updated and more detailed documentation is available [here](#).

The template syntax is a very thin layer around the Python language. It's main purpose is to ensure correct indentation of blocks, so you can format your template without worrying about indentions. It does not prevent your template code from doing bad stuff, so **never ever** execute template code from untrusted sources.

Here is how it works:

- Lines starting with % are interpreted as python code. You can indent these lines but you don't have to. The template engine handles the correct indentation of python blocks.
- A line starting with %end closes a python block opened by %if ..., %for ... or other block statements. Explicitly closing of blocks is required.
- Every other line is just returned as text.
- {{...}} within a text line is replaced by the result of the included python statement. This is used to include template variables.
- The two statements %include and %rebase have special meanings

Here is a simple example, printing a HTML-list of names.

```
1 <ul>
2 % for name in names:
3   <li>{{name}}</li>
4 % end
5 </ul>
6
7 #!python
8 import template
9 print template('mylist', names=['Marc', 'Susan', 'Alice', 'Bob'])
```

You can include other template using the %include statement followed by a template name and an optional argument list. The include-line is replaced by the rendered result of the named sub-template.

```
1 <h1>{{title}}</h1>
2
3 #!html
4 %include header_template title='Hello World'
5 <p>
6   Hello World!
7 </p>
```

The %rebase statement is the inverse of %include and is used to render a template into a surrounding base-template. This is similar to the 'Inheritance' feature found in most other template engines. The base-template can access all the parameters specified by the %rebase statement and use an empty %include statement to include the text returned by the rebased template.

```
1 <h1>{{title}}</h1>
2 <p>
3   %include
4 </p>
5
6 #!html
7 %rebase paragraph_template title='Hello World'
8 hello world!
```

And a last thing: You can add \\ to the end of a text line preceding a line of python code to suppress the line break.

```
1 List: \\
2 %for i in range(5):
```

```
3 {{i}}
4 <br />
5
6 #!html
7 List: 1 2 3 4 5 <br />
```

Thats all.

Key/Value Databases

Warning: The included key/value database is depreciated since 0.6.4.

Please switch to a [real key value database](#).

Using WSGI and Middleware

A call to `bottle.default_app()` returns your WSGI application. After applying as many WSGI middleware modules as you like, you can tell `bottle.run()` to use your wrapped application, instead of the default one.

```
1 from bottle import default_app, run
2 app = default_app()
3 newapp = YourMiddleware(app)
4 run(app=newapp)
```

How default_app() works

Bottle creates a single instance of `bottle.Bottle()` and uses it as a default for most of the modul-level decorators and the `bottle.run()` routine. `bottle.default_app()` returns (or changes) this default. You may, however, create your own instances of `bottle.Bottle()`.

```
1 from bottle import Bottle, run
2 mybottle = Bottle()
3 @mybottle.route('/')
4 def index():
5     return 'default_app'
6 run(app=mybottle)
```

Development

Bottle has two features that may be helpfull during development.

Debug Mode

In debug mode, bottle is much more verbose and tries to help you finding bugs. You should never use debug mode in production environments.

```
1 import bottle
2 bottle.debug(True)
```

This does the following:

- Exceptions will print a stacktrace

- Error pages will contain that stacktrace
- Templates will not be cached.

Auto Reloading

During development, you have to restart the server a lot to test your recent changes. The auto reloader can do this for you. Every time you edit a module file, the reloader restarts the server process and loads the newest version of your code.

```
1 from bottle import run
2 run(reloader=True)
```

How it works: The main process will not start a server, but spawn a new child process using the same command line arguments used to start the main process. All module level code is executed at least twice! Be carefull.

The child process will have `os.environ['BOTTLE_CHILD']` set to `true` and start as a normal non-reloading app server. As soon as any of the loaded modules changes, the child process is terminated and respawned by the main process. Changes in template files will not trigger a reload. Please use debug mode to deactivate template caching.

The reloading depends on the ability to stop the child process. If you are running on Windows or any other operating system not supporting `signal.SIGINT` (which raises `KeyboardInterrupt` in Python), `signal.SIGTERM` is used to kill the child. Note that exit handlers and finally clauses, etc., are not executed after a `SIGTERM`.

Deployment

Bottle uses the build-in `wsgiref.SimpleServer` by default. This non-threading HTTP server is perfectly fine for development and early production, but may become a performance bottleneck when server load increases.

There are three ways to eliminate this bottleneck:

- Use a multi-threaded server adapter
- Spread the load between multiple bottle instances
- Do both

Multi-Threaded Server

The easiest way to increase performance is to install a multi-threaded and WSGI-capable HTTP server like [Paste](#), [flup](#), [cherry.py](#) or [fapws3](#) and use the corresponding bottle server-adapter.

```
1 from bottle import PasteServer, FlupServer, FapwsServer, CherryPyServer
2 bottle.run(server=PasteServer) # Example
```

If bottle is missing an adapter for your favorite server or you want to tweak the server settings, you may want to manually set up your HTTP server and use `bottle.default_app()` to access your WSGI application.

```
1 def run_custom_paste_server(self, host, port):
2     myapp = bottle.default_app()
3     from paste import httpserver
4     httpserver.serve(myapp, host=host, port=port)
```

Multiple Server Processes

A single Python process can only utilise one CPU at a time, even if there are more CPU cores available. The trick is to balance the load between multiple independent Python processes to utilise all of your CPU cores.

Instead of a single Bottle application server, you start one instances of your server for each CPU core available using different local port (localhost:8080, 8081, 8082, ...). Then a high performance load balancer acts as a reverse proxy and forwards each new requests to a random Bottle processes, spreading the load between all available backed server instances. This way you can use all of your CPU cores and even spread out the load between different physical servers.

But there are a few drawbacks:

- You can't easily share data between multiple Python processes.
- It takes a lot of memory to run several copies of Python and Bottle at the same time.

One of the fastest load balancer available is [pound](#) but most common web servers have a proxy-module that can do the work just fine.

I'll add examples for [lighttpd](#) and [Apache](#) web servers soon.

Apache mod_wsgi

Instead of running your own HTTP server from within Bottle, you can attach Bottle applications to an [Apache server](#) using [mod_wsgi](#) and Bottles WSGI interface.

All you need is an `app.wsgi` file that provides an `application` object. This object is used by `mod_wsgi` to start your application and should be a WSGI conform Python callable.

```
1 # File: /var/www/yourapp/app.wsgi
2
3 # Change working directory so relative paths (and template lookup) work again
4 os.chdir(os.path.dirname(__file__))
5
6 import bottle
7 # ... add or import your bottle app code here ...
8 # Do NOT use bottle.run() with mod_wsgi
9 application = bottle.default_app()
```

The Apache configuration may look like this:

```
1 <VirtualHost *>
2     ServerName example.com
3
4     WSGIDaemonProcess yourapp user=www-data group=www-data processes=1 threads=5
5     WSGIScriptAlias / /var/www/yourapp/app.wsgi
6
7     <Directory /var/www/yourapp>
8         WSGIProcessGroup yourapp
9         WSGIApplicationGroup %{GLOBAL}
10        Order deny,allow
11        Allow from all
12    </Directory>
13 </VirtualHost>
```

Google AppEngine

I didn't test this myself but several Bottle users reported that this works just fine.

```
1 import bottle
2 from google.appengine.ext.webapp import util
3 # ... add or import your bottle app code here ...
4 # Do NOT use bottle.run() with AppEngine
5 util.run_wsgi_app(bottle.default_app())
```

Good old CGI

CGI is slow as hell, but it works.

```
1 import bottle
2 # ... add or import your bottle app code here ...
3 bottle.run(server=bottle.CGIServer)
```

Edit this page at [GitHub](#)