



# **Bottle Documentation**

*Release 0.9.8*

**Marcel Hellkamp**

September 24, 2015



<b>1</b>	<b>User's Guide</b>	<b>3</b>
1.1	Documentation . . . . .	3
1.2	SimpleTemplate Engine . . . . .	20
1.3	API Reference . . . . .	23
1.4	List of available Plugins . . . . .	31
<b>2</b>	<b>Knowledge Base</b>	<b>35</b>
2.1	Tutorial: Todo-List Application . . . . .	35
2.2	Primer to Asynchronous Applications . . . . .	49
2.3	Recipes . . . . .	51
2.4	Frequently Asked Questions . . . . .	54
<b>3</b>	<b>Development and Contribution</b>	<b>55</b>
3.1	Release Notes and Changelog . . . . .	55
3.2	Developer Notes . . . . .	57
3.3	Plugin Development Guide . . . . .	60
3.4	List of available Plugins . . . . .	64
<b>4</b>	<b>License</b>	<b>69</b>
	<b>Python Module Index</b>	<b>71</b>



Bottle is a fast, simple and lightweight [WSGI](#) micro web-framework for [Python](#). It is distributed as a single file module and has no dependencies other than the [Python Standard Library](#).

- **Routing:** Requests to function-call mapping with support for clean and dynamic URLs.
- **Templates:** Fast and pythonic *built-in template engine* and support for [mako](#), [jinja2](#) and [cheetah](#) templates.
- **Utilities:** Convenient access to form data, file uploads, cookies, headers and other HTTP related metadata.
- **Server:** Built-in HTTP development server and support for [paste](#), [fapws3](#), [bjoern](#), [Google App Engine](#), [cherrypy](#) or any other [WSGI](#) capable HTTP server.

### Example: “Hello World” in a bottle

```
from bottle import route, run

@route('/:name')
def index(name='World'):
    return '<b>Hello %s!</b>' % name

run(host='localhost', port=8080)
```

### Download and Install

Install the latest stable release via [PyPi](#) (`easy_install -U bottle`) or download [bottle.py](#) (unstable) into your project directory. There are no hard <sup>1</sup> dependencies other than the Python standard library. Bottle runs with **Python 2.5+ and 3.x** (using 2to3)

---

<sup>1</sup> Usage of the template or server adapter classes of course requires the corresponding template or server modules.



---

## User's Guide

---

Start here if you want to learn how to use the bottle framework for web development. If you have any questions not answered here, feel free to ask the [mailing list](#).

### 1.1 Documentation

This tutorial introduces you to the concepts and features of the Bottle web framework. If you have questions not answered here, please check the [Frequently Asked Questions](#) page, file a ticket at the [issue](#) tracker or send an e-mail to the [mailing list](#).

#### A quick overview:

- *Request Routing*: Web development starts with binding URLs to code. This section tells you how to do it.
- *Generating content*: You have to return something to the Browser. Bottle makes it easy for you, supporting more than just plain strings.
- *Accessing Request Data*: Each client request carries a lot of information. HTTP-headers, form data and cookies to name just three. Here is how to use them.
- *Templates*: You don't want to clutter your Python code with HTML fragments, do you? Templates separate code from presentation.
- *Debug Mode*: These tools and features will help you during development.
- *Deployment*: Get it up and running.

#### 1.1.1 Installation

Bottle does not depend on any external libraries. You can just download `bottle.py` into your project directory and start coding:

```
$ curl -O http://bottlepy.org/bottle.py
$ 2to3 -w bottle.py # Python 3.x users only!
```

This will get you the latest development snapshot that includes all the new features. If you prefer a more stable environment, you should stick with a stable release. These are available on [PyPi](#) and can be installed via **pip** (recommended), **easy\_install** or your Linux distributions package manager:

```
$ sudo pip install bottle           # recommended
$ sudo easy_install bottle          # alternative to pip
$ sudo apt-get install python-bottle # works for debian, ubuntu, ...
```

In either way, you'll need Python 2.5 or newer to run bottle applications. If you do not have permissions to install packages system-wide or simply don't want to, I suggest creating a [virtualenv](#) first.

### 1.1.2 A minimal Bottle Application

This tutorial assumes you have Bottle either *installed or copied* into your project directory. Let's start with a very basic "Hello World" example:

```
from bottle import route, run

@route('/hello')
def hello():
    return "Hello World!"

run(host='localhost', port=8080)
```

What's happening here?

1. First we import some Bottle components. The `route()` decorator and the `run()` function.
2. The `route()` decorator is used to bind a piece of code to an URL. In this example we want to answer requests to the `/hello` URL.
3. This function is the *handler function* or *callback* for the `/hello` route. It is called every time someone requests the `/hello` URL and is responsible for generating the page content.
4. In this example we simply return a string to the browser.
5. In the last line we start the actual HTTP server. The default is a development server running on 'localhost' port 8080 and serving requests until you hit `Control-C`.

This is it. Run this script, visit <http://localhost:8080/hello> and you will see "Hello World!" in your browser. Of course this is a very simple example, but it shows the basic concept of how applications are built with Bottle. Continue reading and you'll see what else is possible.

#### The Application Object

For the sake of simplicity, most examples in this tutorial use a module-level `route()` decorator to bind routes. This decorator adds routes to a global application object that is created automatically in the background. If you prefer a more explicit way to define your application and don't mind the extra typing, you can create a separate application object and use that instead of the global one:

```
from bottle import Bottle

app = Bottle()

@app.route('/hello')
def hello():
    return "Hello World!"

run(app, host='localhost', port=8080)
```

The object-oriented approach is further described in the [Default Application](#) section. Just keep in mind that you have a choice.



### 1.1.3 Request Routing

As you have learned before, *routes* are used to map URLs to callback functions. These functions are executed on every request that matches the route and their return value is returned to the browser. You can add any number of routes to a callback using the `route()` decorator.

```
from bottle import route

@route('/')
@route('/index.html')
def index():
    return "<a href='/hello'>Go to Hello World page</a>"

@route('/hello')
def hello():
    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. All URLs not covered by a route are answered with a “404 Page not found” error page.

### Dynamic Routes

Bottle has a special syntax to add wildcards to a route and allow a single route to match a wide range of URLs. These *dynamic routes* are often used by blogs or wikis to create nice looking and meaningful URLs such as `/archive/2010/04/21` or `/wiki/Page_Title`. Why? Because **cool URIs don’t change**. Let’s add a `:name` wildcard to our last example:

```
@route('/hello/:name')
def hello(name):
    return "Hello %s!" % name
```

This dynamic route will match `/hello/alice` as well as `/hello/bob`. Each URL fragment covered by a wildcard is passed to the callback function as a keyword argument so you can use the information in your application.

Normal wildcards match everything up to the next slash. You can add a regular expression to change that:

```
@route('/object/:id#[0-9]+#')
def view_object(id):
    return "Object ID: %d" % int(id)
```

As you can see, the keyword argument contains a string even if the wildcard is configured to only match digits. You have to explicitly cast it into an integer if you need to.

### HTTP Request Methods

The HTTP protocol defines several **request methods** (sometimes referred to as “verbs”) for different tasks. GET is the default for all routes with no other method specified. These routes will match GET requests only. To handle other methods such as POST, PUT or DELETE, you may add a `method` keyword argument to the `route()` decorator or use one of the four alternative decorators: `get()`, `post()`, `put()` or `delete()`.

The POST method is commonly used for HTML form submission. This example shows how to handle a login form using POST:

```
from bottle import get, post, request

#@route('/login')
```

```
@get('/login')
def login_form():
    return '''<form method="POST">
        <input name="name" type="text" />
        <input name="password" type="password" />
    </form>'''

#@route('/login', method='POST')
@post('/login')
def login_submit():
    name = request.forms.get('name')
    password = request.forms.get('password')
    if check_login(name, password):
        return "<p>Your login was correct</p>"
    else:
        return "<p>Login failed</p>"
```

In this example the `/login` URL is bound to two distinct callbacks, one for GET requests and another for POST requests. The first one displays a HTML form to the user. The second callback is invoked on a form submission and checks the login credentials the user entered into the form. The use of `Request.forms` is further described in the [Accessing Request Data](#) section.

### Automatic Fallbacks

The special HEAD method is used to ask for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information about a resource without having to download the entire document. Bottle handles these requests automatically by falling back to the corresponding GET route and cutting off the request body, if present. You don't have to specify any HEAD routes yourself.

Additionally, the non-standard ANY method works as a low priority fallback: Routes that listen to ANY will match requests regardless of their HTTP method but only if no other more specific route is defined. This is helpful for *proxy-routes* that redirect requests to more specific sub-applications.

To sum it up: HEAD requests fall back to GET routes and all requests fall back to ANY routes, but only if there is no matching route for the original request method. It's as simple as that.

### Routing Static Files

Static files such as images or css files are not served automatically. You have to add a route and a callback to control which files get served and where to find them:

```
from bottle import static_file
@route('/static/:filename')
def server_static(filename):
    return static_file(filename, root='/path/to/your/static/files')
```

The `static_file()` function is a helper to serve files in a safe and convenient way (see [Static Files](#)). This example is limited to files directly within the `/path/to/your/static/files` directory because the `:filename` wildcard won't match a path with a slash in it. To serve files in subdirectories too, we can loosen the wildcard a bit:

```
@route('/static/:path#.+#')
def server_static(path):
    return static_file(path, root='/path/to/your/static/files')
```

Be careful when specifying a relative root-path such as `root='./static/files'`. The working directory (`./`) and the project directory are not always the same.

## Error Pages

If anything goes wrong, Bottle displays an informative but fairly boring error page. You can override the default for a specific HTTP status code with the `error()` decorator:

```
@error(404)
def error404(error):
    return 'Nothing here, sorry'
```

From now on, *404 File not Found* errors will display a custom error page to the user. The only parameter passed to the error-handler is an instance of `HTTPError`. Apart from that, an error-handler is quite similar to a regular request callback. You can read from `request`, write to `response` and return any supported data-type except for `HTTPError` instances.

Error handlers are used only if your application returns or raises an `HTTPError` exception (`abort()` does just that). Changing `Request.status` or returning `HTTPResponse` won't trigger the error handler.

### 1.1.4 Generating content

In pure WSGI, the range of types you may return from your application is very limited. Applications must return an iterable yielding byte strings. You may return a string (because strings are iterable) but this causes most servers to transmit your content char by char. Unicode strings are not allowed at all. This is not very practical.

Bottle is much more flexible and supports a wide range of types. It even adds a `Content-Length` header if possible and encodes unicode automatically, so you don't have to. What follows is a list of data types you may return from your application callbacks and a short description of how these are handled by the framework:

**Dictionaries** As mentioned above, Python dictionaries (or subclasses thereof) are automatically transformed into JSON strings and returned to the browser with the `Content-Type` header set to `application/json`. This makes it easy to implement json-based APIs. Data formats other than json are supported too. See the `tutorial-output-filter` to learn more.

**Empty Strings, False, None or other non-true values:** These produce an empty output with `Content-Length` header set to 0.

**Unicode strings** Unicode strings (or iterables yielding unicode strings) are automatically encoded with the codec specified in the `Content-Type` header (`utf8` by default) and then treated as normal byte strings (see below).

**Byte strings** Bottle returns strings as a whole (instead of iterating over each char) and adds a `Content-Length` header based on the string length. Lists of byte strings are joined first. Other iterables yielding byte strings are not joined because they may grow too big to fit into memory. The `Content-Length` header is not set in this case.

**Instances of `HTTPError` or `HTTPResponse`** Returning these has the same effect as when raising them as an exception. In case of an `HTTPError`, the error handler is applied. See [Error Pages](#) for details.

**File objects** Everything that has a `.read()` method is treated as a file or file-like object and passed to the `wsgi.file_wrapper` callable defined by the WSGI server framework. Some WSGI server implementations can make use of optimized system calls (`sendfile`) to transmit files more efficiently. In other cases this just iterates over chunks that fit into memory. Optional headers such as `Content-Length` or `Content-Type` are *not* set automatically. Use `send_file()` if possible. See [Static Files](#) for details.

**Iterables and generators** You are allowed to use `yield` within your callbacks or return an iterable, as long as the iterable yields byte strings, unicode strings, `HTTPError` or `HTTPResponse` instances. Nested iterables are not supported, sorry. Please note that the HTTP status code and the headers are sent to the browser as soon as the iterable yields its first non-empty value. Changing these later has no effect.

The ordering of this list is significant. You may for example return a subclass of `str` with a `read()` method. It is still treated as a string instead of a file, because strings are handled first.

## Changing the Default Encoding

Bottle uses the *charset* parameter of the Content-Type header to decide how to encode unicode strings. This header defaults to `text/html; charset=UTF8` and can be changed using the `Response.content_type` attribute or by setting the `Response.charset` attribute directly. (The `Response` object is described in the section *The Response Object*.)

```
from bottle import response
@route('/iso')
def get_iso():
    response.charset = 'ISO-8859-15'
    return u'This will be sent with ISO-8859-15 encoding.'

@route('/latin9')
def get_latin():
    response.content_type = 'text/html; charset=latin9'
    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` attribute (which is used to encode unicode).

## Static Files

You can directly return file objects, but `static_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 responses (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.

```
from bottle import static_file
@route('/images/:filename#.*\.png#')
def send_image(filename):
    return static_file(filename, root='/path/to/image/files', mimetype='image/png')

@route('/static/:filename')
def send_static(filename):
    return static_file(filename, root='/path/to/static/files')
```

You can raise the return value of `static_file()` as an exception if you really need to.

## Forced Download

Most browsers try to open downloaded files if the MIME type is known and assigned to an application (e.g. PDF files). If this is not what you want, you can force a download-dialog and even suggest a filename to the user:

```
@route('/download/:filename')
def download(filename):
    return static_file(filename, root='/path/to/static/files', download=filename)
```

If the `download` parameter is just `True`, the original filename is used.

## HTTP Errors and Redirects

The `abort()` function is a shortcut for generating HTTP error pages.

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

To redirect a client to a different URL, you can send a 303 See Other response with the Location header set to the new URL. `redirect()` does that for you:

```
from bottle import redirect
@route('/wrong/url')
def wrong():
    redirect("/right/url")
```

You may provide a different HTTP status code as a second parameter.

---

**Note:** Both functions will interrupt your callback code by raising an *HTTPError* exception.

---

## Other Exceptions

All exceptions other than *HTTPResponse* or *HTTPError* will result in a 500 Internal Server Error response, so they won't crash your WSGI server. You can turn off this behaviour to handle exceptions in your middleware by setting `bottle.app().catchall` to `False`.

## The Response Object

Response meta-data such as the HTTP status code, response header and cookies are stored in an object called *response* up to the point where they are transmitted to the browser. You can manipulate these meta-data directly or use the predefined helper methods to do so. The full API and feature list is described in the API section (see *Response*), but the most common use cases and features are covered here, too.

### Status Code

The HTTP status code controls the behaviour of the browser and defaults to 200 OK. In most scenarios you won't need to set the `Response.status` attribute manually, but use the `abort()` helper or return an *HTTPResponse* instance with the appropriate status code. Any integer is allowed but only the codes defined by the HTTP specification will have an effect other than confusing the browser and breaking standards.

### Response Header

Add values to the `Response.headers` dictionary to add or change response headers. Note that the keys are case-insensitive.

```
@route('/wiki/:page')
def wiki(page):
    response.headers['Content-Language'] = 'en'
    return get_wiki_page(page)
```

## Cookies

A cookie is a piece of text stored in the user's browser. You can access cookies via `Request.get_cookie()` and set new cookies with the `Response.set_cookie()` method:

```
@route('/hello')
def hello_again(self):
    if request.get_cookie("visited"):
        return "Welcome back! Nice to see you again"
    else:
        response.set_cookie("visited", "yes")
        return "Hello there! Nice to meet you"
```

But there are some gotchas:

- Cookies are limited to 4kb of text in most browsers.
- Some users configure their browsers to not accept cookies at all. Most search-engines ignore cookies, too. Make sure that your application is still usable without cookies.
- Cookies are stored at client side and not encrypted in any way. Whatever you store in a cookie, the user can read it. Worth than that, an attacker might be able to steal a user's cookies through **XSS** vulnerabilities on your side. Some viruses are known to read the browser cookies, too. Do not store confidential information in cookies, ever.
- Cookies are easily forged by malicious clients. Do not trust cookies.

### Signed Cookies

As mentioned above, cookies are easily forged by malicious clients. Bottle can cryptographically sign your cookies to prevent this kind of manipulation. All you have to do is to provide a signature key whenever you read or set a cookie and keep that key a secret. As a result, `Request.get_cookie()` will return `None` if the cookie is not signed or the signature keys don't match:

```
@route('/login')
def login():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if check_user_credentials(username, password):
        response.set_cookie("account", username, secret='some-secret-key')
        return "Welcome %s! You are now logged in." % username
    else:
        return "Login failed."

@route('/restricted')
def restricted_area(self):
    username = request.get_cookie("account", secret='some-secret-key')
    if username:
        return "Hello %s. Welcome back." % username
    else:
        return "You are not logged in. Access denied."
```

In addition, Bottle automatically pickles and unpickles any data stored to signed cookies. This allows you to store any pickle-able object (not only strings) to cookies, as long as the pickled data does not exceed the 4kb limitation.

**Warning:** Signed cookies are not encrypted (the client can still see the content) and not copy-protected (the client can restore an old cookie). The main intention is to make pickling and unpickling save, not to store secret information at client side.

### 1.1.5 Accessing Request Data

Bottle provides access to HTTP related meta-data such as cookies, headers and POST form data through a global request object. This object always contains information about the *current* request, as long as it is accessed from within a callback function. This works even in multi-threaded environments where multiple requests are handled at the same time. For details on how a global object can be thread-safe, see `contextlocal`.

**Note:** Bottle stores most of the parsed HTTP meta-data in *MultiDict* instances. These behave like normal dictionaries but are able to store multiple values per key. The standard dictionary access methods will only return a single value. Use the `MultiDict.getall()` method to receive a (possibly empty) list of all values for a specific key. The *HeaderDict* class inherits from *MultiDict* and additionally uses case insensitive keys.

The full API and feature list is described in the API section (see *Request*), but the most common use cases and features are covered here, too.

#### HTTP Header

Header are stored in `Request.header`. The attribute is an instance of *HeaderDict* which is basically a dictionary with case-insensitive keys:

```
from bottle import route, request
@route('/is_ajax')
def is_ajax():
    if request.header.get('X-Requested-With') == 'XMLHttpRequest':
        return 'This is an AJAX request'
    else:
        return 'This is a normal request'
```

#### Cookies

Cookies are stored in `Request.COOKIES` as a normal dictionary. The `Request.get_cookie()` method allows access to *Cookies* as described in a separate section. This example shows a simple cookie-based view counter:

```
from bottle import route, request, response
@route('/counter')
def counter():
    count = int( request.COOKIES.get('counter', '0') )
    count += 1
    response.set_cookie('counter', str(count))
    return 'You visited this page %d times' % count
```

#### Query Strings

The query string (as in `/forum?id=1&page=5`) is commonly used to transmit a small number of key/value pairs to the server. You can use the `Request.GET` dictionary to access these values and the `Request.query_string` attribute to get the whole string.

```
from bottle import route, request, response
@route('/forum')
def display_forum():
    forum_id = request.GET.get('id')
    page = request.GET.get('page', '1')
    return 'Forum ID: %s (page %s)' % (forum_id, page)
```

## POST Form Data and File Uploads

The request body of POST and PUT requests may contain form data encoded in various formats. Use the `Request.forms` attribute (a *MultiDict*) to access normal POST form fields. File uploads are stored separately in `Request.files` as `cgi.FieldStorage` instances. The `Request.body` attribute holds a file object with the raw body data.

Here is an example for a simple file upload form:

```
<form action="/upload" method="post" enctype="multipart/form-data">
  <input type="text" name="name" />
  <input type="file" name="data" />
</form>
```

```
from bottle import route, request
@route('/upload', method='POST')
def do_upload():
    name = request.forms.get('name')
    data = request.files.get('data')
    if name and data:
        raw = data.file.read() # This is dangerous for big files
        filename = data.filename
        return "Hello %s! You uploaded %s (%d bytes)." % (name, filename, len(raw))
    return "You missed a field."
```

## WSGI environment

The *Request* object stores the WSGI environment dictionary in `Request.environ` and allows dict-like access to its values. See the [WSGI specification](#) for details.

```
@route('/my_ip')
def show_ip():
    ip = request.environ.get('REMOTE_ADDR')
    # or ip = request.get('REMOTE_ADDR')
    # or ip = request['REMOTE_ADDR']
    return "Your IP is: %s" % ip
```

### 1.1.6 Templates

Bottle comes with a fast and powerful built-in template engine called *SimpleTemplate Engine*. To render a template you can use the `template()` function or the `view()` decorator. All you have to do is to provide the name of the template and the variables you want to pass to the template as keyword arguments. Here's a simple example of how to render a template:

```
@route('/hello')
@route('/hello/:name')
def hello(name='World'):
    return template('hello_template', name=name)
```

This will load the template file `hello_template.tpl` and render it with the `name` variable set. Bottle will look for templates in the `./views/` folder or any folder specified in the `bottle.TEMPLATE_PATH` list.

The `view()` decorator allows you to return a dictionary with the template variables instead of calling `template()`:



```
@route('/hello')
@route('/hello/:name')
@view('hello_template')
def hello(name='World'):
    return dict(name=name)
```

## Syntax

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 indentation. Follow the link for a full syntax description: [SimpleTemplate Engine](#)

Here is an example template:

```
%if name == 'World':
    <h1>Hello {{name}}!</h1>
    <p>This is a test.</p>
%else:
    <h1>Hello {{name.title()}}!</h1>
    <p>How are you?</p>
%end
```

## Caching

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

## 1.1.7 Plugins

New in version 0.9.

Bottle's core features cover most common use-cases, but as a micro-framework it has its limits. This is where “Plugins” come into play. Plugins add missing functionality to the framework, integrate third party libraries, or just automate some repetitive work.

We have a growing [List of available Plugins](#) and most plugins are designed to be portable and re-usable across applications. The chances are high that your problem has already been solved and a ready-to-use plugin exists. If not, the [Plugin Development Guide](#) may help you.

The effects and APIs of plugins are manifold and depend on the specific plugin. The ‘sqlite’ plugin for example detects callbacks that require a db keyword argument and creates a fresh database connection object every time the callback is called. This makes it very convenient to use a database:

```
from bottle import route, install, template
from bottle_sqlite import SQLitePlugin

install(SQLitePlugin(dbfile='/tmp/test.db'))

@route('/show/:post_id')
def show(db, post_id):
    c = db.execute('SELECT title, content FROM posts WHERE id = ?', (int(post_id),))
    row = c.fetchone()
    return template('show_post', title=row['title'], text=row['content'])

@route('/contact')
```

```
def contact_page():
    ''' This callback does not need a db connection. Because the 'db'
        keyword argument is missing, the sqlite plugin ignores this callback
        completely. '''
    return template('contact')
```

Other plugin may populate the thread-safe `local` object, change details of the `request` object, filter the data returned by the callback or bypass the callback completely. An “auth” plugin for example could check for a valid session and return a login page instead of calling the original callback. What happens exactly depends on the plugin.

## Application-wide Installation

Plugins can be installed application-wide or just to some specific routes that need additional functionality. Most plugins are save to be installed to all routes and are smart enough to not add overhead to callbacks that do not need their functionality.

Let us take the ‘sqlite’ plugin for example. It only affects route callbacks that need a database connection. Other routes are left alone. Because of this, we can install the plugin application-wide with no additional overhead.

To install a plugin, just call `install()` with the plugin as first argument:

```
from bottle_sqlite import SQLitePlugin
install(SQLitePlugin(dbfile='/tmp/test.db'))
```

The plugin is not applied to the route callbacks yet. This is delayed to make sure no routes are missed. You can install plugins first and add routes later, if you want to. The order of installed plugins is significant, though. If a plugin requires a database connection, you need to install the database plugin first.

## Uninstall Plugins

You can use a name, class or instance to `uninstall()` a previously installed plugin:

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test.db')
install(sqlite_plugin)

uninstall(sqlite_plugin) # uninstall a specific plugin
uninstall(SQLitePlugin) # uninstall all plugins of that type
uninstall('sqlite')     # uninstall all plugins with that name
uninstall(True)         # uninstall all plugins at once
```

Plugins can be installed and removed at any time, even at runtime while serving requests. This enables some neat tricks (installing slow debugging or profiling plugins only when needed) but should not be overused. Each time the list of plugins changes, the route cache is flushed and all plugins are re-applied.

---

**Note:** The module-level `install()` and `uninstall()` functions affect the *Default Application*. To manage plugins for a specific application, use the corresponding methods on the *Bottle* application object.

---

## Route-specific Installation

The `apply` parameter of the `route()` decorator comes in handy if you want to install plugins to only a small number of routes:

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test.db')

@route('/create', apply=[sqlite_plugin])
def create(db):
    db.execute('INSERT INTO ...')
```

## Blacklisting Plugins

You may want to explicitly disable a plugin for a number of routes. The `route()` decorator has a `skip` parameter for this purpose:

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test.db')
install(sqlite_plugin)

@route('/open/:db', skip=[sqlite_plugin])
def open_db(db):
    # The 'db' keyword argument is not touched by the plugin this time.
    if db in ('test', 'test2'):
        # The plugin handle can be used for runtime configuration, too.
        sqlite_plugin.dbfile = '/tmp/%s.db' % db
        return "Database File switched to: /tmp/%s.db" % db
    abort(404, "No such database.")
```

The `skip` parameter accepts a single value or a list of values. You can use a name, class or instance to identify the plugin that is to be skipped. Set `skip=True` to skip all plugins at once.

## Plugins and Sub-Applications

Most plugins are specific to the application they were installed to. Consequently, they should not affect sub-applications mounted with `Bottle.mount()`. Here is an example:

```
root = Bottle()
root.mount(apps.blog, '/blog')

@root.route('/contact', template='contact')
def contact():
    return {'email': 'contact@example.com'}

root.install(plUGINS.WTForms())
```

Whenever you mount an application, Bottle creates a proxy-route on the main-application that relays all requests to the sub-application. Plugins are disabled for this kind of proxy-routes by default. As a result, our (fictional) `WTForms` plugin affects the `/contact` route, but does not affect the routes of the `/blog` sub-application.

This behavior is intended as a sane default, but can be overridden. The following example re-activates all plugins for a specific proxy-route:

```
root.mount(apps.blog, '/blog', skip=None)
```

But there is a snag: The plugin sees the whole sub-application as a single route, namely the proxy-route mentioned above. In order to affect each individual route of the sub-application, you have to install the plugin to the application explicitly.

## 1.1.8 Development

You learned the basics and want to write your own application? Here are some tips that might help you to be more productive.

### Default Application

Bottle maintains a global stack of *Bottle* instances and uses the top of the stack as a default for some of the module-level functions and decorators. The `route()` decorator, for example, is a shortcut for calling `Bottle.route()` on the default application:

```
@route('/')
def hello():
    return 'Hello World'
```

This is very convenient for small applications and saves you some typing, but also means that, as soon as your module is imported, routes are installed to the global application. To avoid this kind of import side-effects, Bottle offers a second, more explicit way to build applications:

```
app = Bottle()

@app.route('/')
def hello():
    return 'Hello World'
```

Separating the application object improves re-usability a lot, too. Other developers can safely import the `app` object from your module and use `Bottle.mount()` to merge applications together.

As an alternative, you can make use of the application stack to isolate your routes while still using the convenient shortcuts:

```
default_app.push()

@route('/')
def hello():
    return 'Hello World'

app = default_app.pop()
```

Both `app()` and `default_app()` are instance of *AppStack* and implement a stack-like API. You can push and pop applications from and to the stack as needed. This also helps if you want to import a third party module that does not offer a separate application object:

```
default_app.push()

import some.module

app = default_app.pop()
```

### Debug Mode

During early development, the debug mode can be very helpful.

```
bottle.debug(True)
```

In this mode, Bottle is much more verbose and provides helpful debugging information whenever an error occurs. It also disables some optimisations that might get in your way and adds some checks that warn you about possible misconfiguration.

Here is an incomplete list of things that change in debug mode:

- The default error page shows a traceback.
- Templates are not cached.
- Plugins are applied immediately.

Just make sure to not use the debug mode on a production server.

## 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.

```
from bottle import run
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 careful.

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 re-spawned 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`.

## 1.1.9 Deployment

Bottle runs on the built-in `wsgiref.WSGIServer` 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 or asynchronous HTTP server.
- Spread the load between multiple Bottle instances.
- Do both.

### Multi-Threaded Server

The easiest way to increase performance is to install a multi-threaded or asynchronous WSGI server like `paste` or `cherry.py` and tell Bottle to start it instead of the default single-threaded one:

```
bottle.run(server='paste') # Example
```

Bottle ships with a lot of ready-to-use adapters for the most common WSGI servers and automates the setup process. Here is an incomplete list:

Name	Homepage	Description
cgi		Run as CGI script
flup	<a href="#">flup</a>	Run as Fast CGI process
gae	<a href="#">gae</a>	Helper for Google App Engine deployments
wsgiref	<a href="#">wsgiref</a>	Single-threaded default server
cherrypy	<a href="#">cherrypy</a>	Multi-threaded and very stable
paste	<a href="#">paste</a>	Multi-threaded, stable, tried and tested
rocket	<a href="#">rocket</a>	Multi-threaded
gunicorn	<a href="#">gunicorn</a>	Pre-forked, partly written in C
fapws3	<a href="#">fapws3</a>	Asynchronous, written in C
tornado	<a href="#">tornado</a>	Asynchronous, powers some parts of Facebook
twisted	<a href="#">twisted</a>	Asynchronous, well tested
diesel	<a href="#">diesel</a>	Asynchronous, based on greenlet
meinheld	<a href="#">meinheld</a>	Asynchronous, partly written in C
bjoern	<a href="#">bjoern</a>	Asynchronous, very fast and written in C
auto		Automatically selects an available server adapter

The full list is available through `server_names`.

If there is no adapter for your favorite server or if you need more control over the server setup, you may want to start the server manually. Refer to the server documentation on how to mount WSGI applications. Here is an example for `paste`:

```
from paste import httpserver
httpserver.serve(bottle.default_app(), host='0.0.0.0', port=80)
```

## 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 utilize all of your CPU cores.

Instead of a single Bottle application server, you start one instance 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 back end server instances. This way you can use all of your CPU cores and even spread out the load between different physical servers.

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

## 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 Bottle's 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-compatible Python callable.

File `/var/www/yourapp/app.wsgi`:

```
# Change working directory so relative paths (and template lookup) work again
os.chdir(os.path.dirname(__file__))

import bottle
# ... build or import your bottle application here ...
# Do NOT use bottle.run() with mod_wsgi
application = bottle.default_app()
```

The Apache configuration may look like this:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess yourapp user=www-data group=www-data processes=1 threads=5
    WSGIScriptAlias / /var/www/yourapp/app.wsgi

    <Directory /var/www/yourapp>
        WSGIProcessGroup yourapp
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

## Google AppEngine

New in version 0.9.

The `gae` adapter completely automates the Google App Engine deployment. It even ensures that a `main()` function is present in your `__main__` module to enable [App Caching](#) (which drastically improves performance):

```
import bottle
# ... build or import your bottle application here ...
bottle.run(server='gae')
```

It is always a good idea to let GAE serve static files directly. Here is example `app.yaml`:

```
application: myapp
version: 1
runtime: python
api_version: 1

handlers:
- url: /static
  static_dir: static

- url: /.*
  script: myapp.py
```

## Good old CGI

CGI is slow as hell, but it works:

```
import bottle
# ... build or import your bottle application here ...
bottle.run(server=bottle.CGIServer)
```

### 1.1.10 Glossary

**callback** Programmer code that is to be called when some external action happens. In the context of web frameworks, the mapping between URL paths and application code is often achieved by specifying a callback function for each URL.

**decorator** A function returning another function, usually applied as a function transformation using the `@decorator` syntax. See [python documentation for function definition](#) for more about decorators.

**environ** A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

**handler function** A function to handle some specific event or situation. In a web framework, the application is developed by attaching a handler function as callback for each specific URL comprising the application.

**source directory** The directory which, including its subdirectories, contains all source files for one Sphinx project.

## 1.2 SimpleTemplate Engine

Bottle comes with a fast, powerful and easy to learn built-in template engine called *SimpleTemplate* or *stpl* for short. It is the default engine used by the `view()` and `template()` helpers but can be used as a stand-alone general purpose template engine too. This document explains the template syntax and shows examples for common use cases.

### Basic API Usage:

*SimpleTemplate* implements the *BaseTemplate* API:

```
>>> from bottle import SimpleTemplate
>>> tpl = SimpleTemplate('Hello {{name}}!')
>>> tpl.render(name='World')
u'Hello World!'
```

In this document we use the `template()` helper in examples for the sake of simplicity:

```
>>> from bottle import template
>>> template('Hello {{name}}!', name='World')
u'Hello World!'
```

Just keep in mind that compiling and rendering templates are two different actions, even if the `template()` helper hides this fact. Templates are usually compiled only once and cached internally, but rendered many times with different keyword arguments.

### 1.2.1 SimpleTemplate Syntax

Python is a very powerful language but its whitespace-aware syntax makes it difficult to use as a template language. *SimpleTemplate* removes some of these restrictions and allows you to write clean, readable and maintainable templates while preserving full access to the features, libraries and speed of the Python language.

**Warning:** The *SimpleTemplate* syntax compiles directly to python bytecode and is executed on each *SimpleTemplate.render()* call. Do not render untrusted templates! They may contain and execute harmful python code.

#### Inline Statements

You already learned the use of the `{{ . . . }}` statement from the “Hello World!” example above, but there is more: any python statement is allowed within the curly brackets as long as it returns a string or something that has a string representation:



```
>>> template('Hello {{name}}!', name='World')
u'Hello World!'
>>> template('Hello {{name.title() if name else "stranger"}}!', name=None)
u'Hello stranger!'
>>> template('Hello {{name.title() if name else "stranger"}}!', name='mArC')
u'Hello Marc!'
```

The contained python statement is executed at render-time and has access to all keyword arguments passed to the `SimpleTemplate.render()` method. HTML special characters are escaped automatically to prevent XSS attacks. You can start the statement with an exclamation mark to disable escaping for that statement:

```
>>> template('Hello {{name}}!', name='<b>World</b>')
u'Hello &lt;b&gt;World&lt;/b&gt;!'
>>> template('Hello {{!name}}!', name='<b>World</b>')
u'Hello <b>World</b>!'
```

### Embedded python code

The `%` character marks a line of python code. The only difference between this and real python code is that you have to explicitly close blocks with an `%end` statement. In return you can align the code with the surrounding template and don't have to worry about correct indentation of blocks. The *SimpleTemplate* parser handles that for you. Lines *not* starting with a `%` are rendered as text as usual:

```
%if name:
    Hi <b>{{name}}</b>
%else:
    <i>Hello stranger</i>
%end
```

The `%` character is only recognised if it is the first non-whitespace character in a line. To escape a leading `%` you can add a second one. `%%` is replaced by a single `%` in the resulting template:

```
This line contains a % but no python code.
%% This text-line starts with '%'
%%% This text-line starts with '%%'
```

### Suppressing line breaks

You can suppress the line break in front of a code-line by adding a double backslash at the end of the line:

```
<span>\\
%if True:
nobreak\\
%end
</span>
```

This template produces the following output:

```
<span>nobreak</span>
```

### The `%include` Statement

You can include other templates using the `%include sub_template [kwargs]` statement. The `sub_template` parameter specifies the name or path of the template to be included. The rest of the line is interpreted as a comma-separated list of `key=statement` pairs similar to keyword arguments in function calls. They

are passed to the sub-template analogous to a `SimpleTemplate.render()` call. The `**kwargs` syntax for passing a dict is allowed too:

```
%include header_template title='Hello World'
<p>Hello World</p>
%include footer_template
```

### The `%rebase` Statement

The `%rebase base_template [kwargs]` statement causes `base_template` to be rendered instead of the original template. The base-template then includes the original template using an empty `%include` statement and has access to all variables specified by `kwargs`. This way it is possible to wrap a template with another template or to simulate the inheritance feature found in some other template engines.

Let's say you have a content template and want to wrap it with a common HTML layout frame. Instead of including several header and footer templates, you can use a single base-template to render the layout frame.

Base-template named `layout.tpl`:

```
<html>
<head>
  <title>{{title or 'No title'}}</title>
</head>
<body>
  %include
</body>
</html>
```

Main-template named `content.tpl`:

```
This is the page content: {{content}}
%rebase layout title='Content Title'
```

Now you can render `content.tpl`:

```
>>> print template('content', content='Hello World!')
```

```
<html>
<head>
  <title>Content Title</title>
</head>
<body>
  This is the page content: Hello World!
</body>
</html>
```

A more complex scenario involves chained rebases and multiple content blocks. The `block_content.tpl` template defines two functions and passes them to a `columns.tpl` base template:

```
%def leftblock():
    Left block content.
%end
%def rightblock():
    Right block content.
%end
%rebase columns left=leftblock, right=rightblock, title=title
```

The `columns.tpl` base-template uses the two callables to render the content of the left and right column. It then wraps itself with the `layout.tpl` template defined earlier:

```
%rebase layout title=title
<div style="width: 50%; float:left">
    %leftblock()
</div>
<div style="width: 50%; float:right">
    %rightblock()
</div>
```

Lets see how `block_content.tpl` renders:

```
>>> print template('block_content', title='Hello World!')
```

```
<html>
<head>
  <title>Hello World</title>
</head>
<body>
<div style="width: 50%; float:left">
  Left block content.
</div>
<div style="width: 50%; float:right">
  Right block content.
</div>
</body>
</html>
```

### 1.2.2 SimpleTemplate API

**class SimpleTemplate** (*source=None, name=None, lookup=[], encoding='utf8', \*\*settings*)

**classmethod split\_comment** (*code*)  
Removes comments (`#...`) from python code.

**render** (*\*args, \*\*kwargs*)  
Render the template using keyword arguments as local variables.

### 1.2.3 Known bugs

Some syntax constructions allowed in python are problematic within a template. The following syntaxes won't work with SimpleTemplate:

- Multi-line statements must end with a backslash (`\`) and a comment, if present, must not contain any additional `#` characters.
- Multi-line strings are not supported yet.

## 1.3 API Reference

This is a mostly auto-generated API. If you are new to bottle, you might find the narrative [Documentation](#) more helpful.

### 1.3.1 Module Contents

The module defines several functions, constants, and an exception.

**debug** (*mode=True*)

Change the debug level. There is only one debug level supported at the moment.

**run** (*app=None*, *server='wsgiref'*, *host='127.0.0.1'*, *port=8080*, *interval=1*, *reloader=False*, *quiet=False*, *\*\*kwargs*)

Start a server instance. This method blocks until the server terminates.

#### Parameters

- **app** – WSGI application or target string supported by `load_app()`. (default: `default_app()`)
- **server** – Server adapter to use. See `server_names` keys for valid names or pass a `ServerAdapter` subclass. (default: `wsgiref`)
- **host** – Server address to bind to. Pass `0.0.0.0` to listens on all interfaces including the external one. (default: `127.0.0.1`)
- **port** – Server port to bind to. Values below 1024 require root privileges. (default: `8080`)
- **reloader** – Start auto-reloading server? (default: `False`)
- **interval** – Auto-reloader interval in seconds (default: `1`)
- **quiet** – Suppress output to stdout and stderr? (default: `False`)
- **options** – Options passed to the server adapter.

**load\_app** (*target*)

Load a bottle application based on a target string and return the application object.

If the target is an import path (e.g. `package.module`), the application stack is used to isolate the routes defined in that module. If the target contains a colon (e.g. `package.module:myapp`) the module variable specified after the colon is returned instead.

**request** = <bottle.Request object>

A thread-safe instance of `Request` representing the *current* request.

**response** = <bottle.Response object>

A thread-safe instance of `Response` used to build the HTTP response.

**HTTP\_CODES** = {200: 'OK', 201: 'Created', 202: 'Accepted', 203: 'Non-Authoritative Information', 204: 'No Content', 205: 'Reset Content', 206: 'Partial Content', 300: 'Multiple Choices', 301: 'Moved Permanently', 302: 'Found', 303: 'See other', 304: 'Not Modified', 305: 'Use proxy', 307: 'Temporary Redirect', 400: 'Bad Request', 401: 'Unauthorized', 402: 'Payment Required', 403: 'Forbidden', 404: 'Not Found', 405: 'Method Not Allowed', 406: 'Not Acceptable', 407: 'Proxy Authentication Required', 408: 'Request Timeout', 409: 'Conflict', 410: 'Gone', 411: 'Length Required', 412: 'Precondition Failed', 413: 'Request Entity Too Large', 414: 'Request-URI Too Long', 415: 'Unsupported Media Type', 416: 'Requested Range Not Satisfiable', 417: 'Expectation Failed', 422: 'Unprocessable Entity', 423: 'Locked', 424: 'Failed Dependency', 426: 'Upgrade Required', 500: 'Internal Server Error', 501: 'Not Implemented', 502: 'Bad Gateway', 503: 'Service Unavailable', 504: 'Gateway Timeout', 505: 'HTTP Version Not Supported', 506: 'Variant Also Negotiates', 507: 'Insufficient Storage', 508: 'Loop Detected', 510: 'Not Extended', 511: 'Network Authentication Required'}

A dict to map HTTP status codes (e.g. 404) to phrases (e.g. 'Not Found')

**app** ()

**default\_app** ()

Return the current *Default Application*. Actually, these are callable instances of `AppStack` and implement a stack-like API.

**class AppStack**

A stack-like list. Calling it returns the head of the stack.

**pop** ()

Return the current default application and remove it from the stack.

**push** (*value=None*)

Add a new *Bottle* instance to the stack

## Routing

Bottle maintains a stack of *Bottle* instances (see *app()* and *AppStack*) and uses the top of the stack as a *default application* for some of the module-level functions and decorators.

**route** (*path*, *method*='GET', *callback*=None, *\*\*options*)

**get** (...)

**post** (...)

**put** (...)

**delete** (...)

Decorator to install a route to the current default application. See *Bottle.route()* for details.

**error** (...)

Decorator to install an error handler to the current default application. See *Bottle.error()* for details.

## WSGI and HTTP Utilities

**parse\_date** (*ims*)

Parse rfc1123, rfc850 and asctime timestamps and return UTC epoch.

**parse\_auth** (*header*)

Parse rfc2617 HTTP authentication header string (basic) and return (user,pass) tuple or None

**cookie\_encode** (*data*, *key*)

Encode and sign a pickle-able object. Return a (byte) string

**cookie\_decode** (*data*, *key*)

Verify and decode an encoded string. Return an object or None.

**cookie\_is\_encoded** (*data*)

Return True if the argument looks like a encoded cookie.

**yieldroutes** (*func*)

Return a generator for routes that match the signature (name, args) of the func parameter. This may yield more than one route if the function takes optional keyword arguments. The output is best described by example:

```
a()          -> '/a'
b(x, y)      -> '/b/:x/:y'
c(x, y=5)    -> '/c/:x' and '/c/:x/:y'
d(x=5, y=6)  -> '/d' and '/d/:x' and '/d/:x/:y'
```

**path\_shift** (*script\_name*, *path\_info*, *shift*=1)

Shift path fragments from PATH\_INFO to SCRIPT\_NAME and vice versa.

**Returns** The modified paths.

**Parameters**

- **script\_name** – The SCRIPT\_NAME path.
- **script\_name** – The PATH\_INFO path.
- **shift** – The number of path fragments to shift. May be negative to change the shift direction. (default: 1)

## Data Structures

**class MultiDict** (*\*a*, *\*\*k*)

A dict that remembers old values for each key

**class HeaderDict** (\*a, \*\*k)

Same as *MultiDict*, but title()s the keys and overwrites.

**class WSGIHeaderDict** (environ)

This dict-like class wraps a WSGI environ dict and provides convenient access to HTTP\_\* fields. Keys and values are native strings (2.x bytes or 3.x unicode) and keys are case-insensitive. If the WSGI environment contains non-native string values, these are de- or encoded using a lossless 'latin1' character set.

The API will remain stable even on changes to the relevant PEPs. Currently PEP 333, 444 and 3333 are supported. (PEP 444 is the only one that uses non-native strings.)

**cgikkeys** = ('CONTENT\_TYPE', 'CONTENT\_LENGTH')

List of keys that do not have a 'HTTP\_' prefix.

**raw** (key, default=None)

Return the header value as is (may be bytes or unicode).

**class AppStack**

A stack-like list. Calling it returns the head of the stack.

**push** (value=None)

Add a new *Bottle* instance to the stack

## Exceptions

**exception BottleException**

A base class for exceptions used by bottle.

**exception HTTPResponse** (output='', status=200, header=None)

Used to break execution and immediately finish the response

**exception HTTPError** (code=500, output='Unknown Error', exception=None, traceback=None, header=None)

Used to generate an error page

**exception RouteReset**

If raised by a plugin or request handler, the route is reset and all plugins are re-applied.

## 1.3.2 The Bottle Class

**class Bottle** (catchall=True, autojson=True, config=None)

WSGI application

**catchall** = None

If true, most exceptions are caught and returned as *HTTPError*

**mount** (app, prefix, \*\*options)

Mount an application to a specific URL prefix. The prefix is added to SCRIPT\_PATH and removed from PATH\_INFO before the sub-application is called.

### Parameters

- **app** – an instance of *Bottle*.
- **prefix** – path prefix used as a mount-point.

All other parameters are passed to the underlying *route()* call.

**install** (plugin)

Add a plugin to the list of plugins and prepare it for being applied to all routes of this application. A plugin may be a simple decorator or an object that implements the *Plugin* API.

**uninstall** (*plugin*)

Uninstall plugins. Pass an instance to remove a specific plugin. Pass a type object to remove all plugins that match that type. Subclasses are not removed. Pass a string to remove all plugins with a matching name attribute. Pass `True` to remove all plugins. The list of affected plugins is returned.

**reset** (*id=None*)

Reset all routes (force plugins to be re-applied) and clear all caches. If an ID is given, only that specific route is affected.

**close** ()

Close the application and all installed plugins.

**match** (*environ*)

(deprecated) Search for a matching route and return a (callback, urlargs) tuple. The first element is the associated route callback with plugins applied. The second value is a dictionary with parameters extracted from the URL. The Router raises `HTTPError` (404/405) on a non-match.

**get\_url** (*routename, \*\*kargs*)

Return a string that matches a named route

**route** (*path=None, method='GET', callback=None, name=None, apply=None, skip=None, \*\*config*)

A decorator to bind a function to a request URL. Example:

```
@app.route('/hello/:name')
def hello(name):
    return 'Hello %s' % name
```

The `:name` part is a wildcard. See `Router` for syntax details.

**Parameters**

- **path** – Request path or a list of paths to listen to. If no path is specified, it is automatically generated from the signature of the function.
- **method** – HTTP method (`GET`, `POST`, `PUT`, ...) or a list of methods to listen to. (default: `GET`)
- **callback** – An optional shortcut to avoid the decorator syntax. `route(..., callback=func)` equals `route(...)(func)`
- **name** – The name for this route. (default: `None`)
- **apply** – A decorator or plugin or a list of plugins. These are applied to the route callback in addition to installed plugins.
- **skip** – A list of plugins, plugin classes or names. Matching plugins are not installed to this route. `True` skips all.

Any additional keyword arguments are stored as route-specific configuration and passed to plugins (see `Plugin.apply()`).

**get** (*path=None, method='GET', \*\*options*)

Equals `route()`.

**post** (*path=None, method='POST', \*\*options*)

Equals `route()` with a `POST` method parameter.

**put** (*path=None, method='PUT', \*\*options*)

Equals `route()` with a `PUT` method parameter.

**delete** (*path=None, method='DELETE', \*\*options*)

Equals `route()` with a `DELETE` method parameter.

**error** (*code=500*)

Decorator: Register an output handler for a HTTP error code

**hook** (*name*)

Return a decorator that attaches a callback to a hook.

**handle** (*path, method='GET'*)

(deprecated) Execute the first matching route callback and return the result. *HTTPResponse* exceptions are caught and returned. If *Bottle.catchall* is true, other exceptions are caught as well and returned as *HTTPError* instances (500).

**wsgi** (*environ, start\_response*)

The bottle WSGI-interface.

### 1.3.3 HTTP Request and Response objects

The *Request* class wraps a WSGI environment and provides helpful methods to parse and access form data, cookies, file uploads and other metadata. Most of the attributes are read-only.

The *Response* class on the other hand stores header and cookie data that is to be sent to the client.

---

**Note:** You usually don't instantiate *Request* or *Response* yourself, but use the module-level instances *bottle.request* and *bottle.response* only. These hold the context for the current request cycle and are updated on every request. Their attributes are thread-local, so it is safe to use the global instance in multi-threaded environments too.

---

**class Request** (*environ=None*)

Represents a single HTTP request using thread-local attributes. The Request object wraps a WSGI environment and can be used as such.

**bind** (*environ*)

Bind a new WSGI environment.

This is done automatically for the global *bottle.request* instance on every request.

**copy** ()

Returns a copy of self

**path\_shift** (*shift=1*)

Shift path fragments from PATH\_INFO to SCRIPT\_NAME and vice versa.

**Parameters** *shift* – The number of path fragments to shift. May be negative to change the shift direction. (default: 1)

**urlparts**

Return a *urlparse.SplitResult* tuple that can be used to reconstruct the full URL as requested by the client. The tuple contains: (scheme, host, path, query\_string, fragment). The fragment is always empty because it is not visible to the server.

**url**

Full URL as requested by the client.

**fullpath**

Request path including SCRIPT\_NAME (if present).

**query\_string**

The part of the URL following the '?'.

**content\_length**

Content-Length header as an integer, -1 if not specified



**headers**

Request HTTP Headers stored in a *HeaderDict*.

**GET**

The QUERY\_STRING parsed into an instance of *MultiDict*.

**POST**

The combined values from *forms* and *files*. Values are either strings (form values) or instances of *cgi.FieldStorage* (file uploads).

**forms**

POST form values parsed into an instance of *MultiDict*.

This property contains form values parsed from an *url-encoded* or *multipart/form-data* encoded POST request body. The values are native strings.

**files**

File uploads parsed into an instance of *MultiDict*.

This property contains file uploads parsed from an *multipart/form-data* encoded POST request body. The values are instances of *cgi.FieldStorage*.

**params**

A combined *MultiDict* with values from *forms* and *GET*. File-uploads are not included.

**auth**

HTTP authorization data as a (user, passwd) tuple. (experimental)

This implementation currently only supports basic auth and returns None on errors.

**COOKIES**

Cookies parsed into a dictionary. Signed cookies are NOT decoded automatically. See *get\_cookie()* for details.

**get\_cookie** (*key*, *secret=None*)

Return the content of a cookie. To read a *Signed Cookies*, use the same *secret* as used to create the cookie (see *Response.set\_cookie()*). If anything goes wrong, None is returned.

**is\_ajax**

True if the request was generated using XMLHttpRequest

**class Response**

Represents a single HTTP response using thread-local attributes.

**bind** ()

Resets the Response object to its factory defaults.

**copy** ()

Returns a copy of self.

**wsgiheader** ()

Returns a wsgi conform list of header/value pairs.

**headerlist**

Returns a wsgi conform list of header/value pairs.

**charset**

Return the charset specified in the content-type header.

This defaults to *UTF-8*.

**COOKIES**

A dict-like SimpleCookie instance. Use *set\_cookie()* instead.

**set\_cookie** (*key*, *value*, *secret=None*, *\*\*kargs*)

Add a cookie or overwrite an old one. If the *secret* parameter is set, create a *Signed Cookie* (described below).

#### Parameters

- **key** – the name of the cookie.
- **value** – the value of the cookie.
- **secret** – required for signed cookies. (default: None)
- **max\_age** – maximum age in seconds. (default: None)
- **expires** – a datetime object or UNIX timestamp. (default: None)
- **domain** – the domain that is allowed to read the cookie. (default: current domain)
- **path** – limits the cookie to a given path (default: /)

If neither *expires* nor *max\_age* are set (default), the cookie lasts only as long as the browser is not closed.

Signed cookies may store any pickle-able object and are cryptographically signed to prevent manipulation. Keep in mind that cookies are limited to 4kb in most browsers.

Warning: Signed cookies are not encrypted (the client can still see the content) and not copy-protected (the client can restore an old cookie). The main intention is to make pickling and unpickling save, not to store secret information at client side.

**delete\_cookie** (*key*, *\*\*kwargs*)

Delete a cookie. Be sure to use the same *domain* and *path* parameters as used to create the cookie.

**get\_content\_type** ()

Current 'Content-Type' header.

**content\_type**

Current 'Content-Type' header.

## 1.3.4 Templates

All template engines supported by *bottle* implement the *BaseTemplate* API. This way it is possible to switch and mix template engines without changing the application code at all.

**class BaseTemplate** (*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, *\*\*settings*)

Base class and minimal API for template adapters

**\_\_init\_\_** (*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, *\*\*settings*)

Create a new template. If the *source* parameter (str or buffer) is missing, the *name* argument is used to guess a template filename. Subclasses can assume that *self.source* and/or *self.filename* are set. Both are strings. The *lookup*, *encoding* and *settings* parameters are stored as instance variables. The *lookup* parameter stores a list containing directory paths. The *encoding* parameter should be used to decode byte strings or files. The *settings* parameter contains a dict for engine-specific settings.

**classmethod search** (*name*, *lookup=[]*)

Search *name* in all directories specified in *lookup*. First without, then with common extensions. Return first hit.

**classmethod global\_config** (*key*, *\*args*)

This reads or sets the global settings stored in *class.settings*.

**prepare** (*\*\*options*)

Run preparations (parsing, caching, ...). It should be possible to call this again to refresh a template or to update settings.

**render** (\*args, \*\*kwargs)

Render the template with the specified local variables and return a single byte or unicode string. If it is a byte string, the encoding must match `self.encoding`. This method must be thread-safe! Local variables may be provided in dictionaries (\*args) or directly, as keywords (\*\*kwargs).

**view** (tpl\_name, \*\*defaults)

Decorator: renders a template for a handler. The handler can control its behavior like that:

- return a dict of template vars to fill out the template
- return something other than a dict and the view decorator will not process the template, but return the handler result as is. This includes returning a `HTTPResponse(dict)` to get, for instance, JSON with `autojson` or other castfilters.

**template** (\*args, \*\*kwargs)

Get a rendered template as a string iterator. You can use a name, a filename or a template string as first parameter. Template rendering arguments can be passed as dictionaries or directly (as keyword arguments).

You can write your own adapter for your favourite template engine or use one of the predefined adapters. Currently there are four fully supported template engines:

Class	URL	Decorator	Render function
<i>SimpleTemplate</i>	<a href="#">SimpleTemplate Engine</a>	<i>view()</i>	<i>template()</i>
MakoTemplate	<a href="http://www.makotemplates.org">http://www.makotemplates.org</a>	<code>mako_view()</code>	<code>mako_template()</code>
CheetahTemplate	<a href="http://www.cheetahtemplate.org/">http://www.cheetahtemplate.org/</a>	<code>cheetah_view()</code>	<code>cheetah_template()</code>
Jinja2Template	<a href="http://jinja.pocoo.org/">http://jinja.pocoo.org/</a>	<code>jinja2_view()</code>	<code>jinja2_template()</code>

To use MakoTemplate as your default template engine, just import its specialised decorator and render function:

```
from bottle import mako_view as view, mako_template as template
```

## 1.4 List of available Plugins

The plugin API is extremely flexible and most plugins are designed to be portable and re-usable across applications. Below is an incomplete list of available plugins, all tested with the latest version of bottle and ready to use.

Have a look at *Plugins* for general questions about plugins (installation, usage). If you plan to develop a new plugin, the *Plugin Development Guide* may help you.

- **Bottle-SQLite**: Adds support for *SQLite* databases.
- **Bottle-Werkzeug**: Integrates the *werkzeug* library (alternative request and response objects, advanced debugging middleware and more).
- **profile**: This plugin collects profiling data and displays it in the browser.

### 1.4.1 Bottle-SQLite

SQLite is a self-contained SQL database engine that runs locally and does not require any additional server software or setup. The `sqlite3` module is part of the Python standard library and already installed on most systems. It is very useful for prototyping database-driven applications that are later ported to larger databases such as PostgreSQL or MySQL.

This plugin simplifies the use of sqlite databases in your Bottle applications. Once installed, all you have to do is to add a `db` keyword argument (configurable) to route callbacks that need a database connection.

## Installation

Install with one of the following commands:

```
$ pip install bottle-sqlite
$ easy_install bottle-sqlite
```

or download the latest version from github:

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/sqlite
$ python setup.py install
```

## Usage

Once installed to an application, the plugin passes an open `sqlite3.Connection` instance to all routes that require a `db` keyword argument:

```
import bottle

app = bottle.Bottle()
plugin = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
app.install(plugin)

@app.route('/show/:item')
def show(item, db):
    row = db.execute('SELECT * from items where name=?', item).fetchone()
    if row:
        return template('showitem', page=row)
    return HTTPError(404, "Page not found")
```

Routes that do not expect a `db` keyword argument are not affected.

The connection handle is configured so that `sqlite3.Row` objects can be accessed both by index (like tuples) and case-insensitively by name. At the end of the request cycle, outstanding transactions are committed and the connection is closed automatically. If an error occurs, any changes to the database since the last commit are rolled back to keep the database in a consistent state.

## Configuration

The following configuration options exist for the plugin class:

- **dbfile**: Database filename (default: in-memory database).
- **keyword**: The keyword argument name that triggers the plugin (default: 'db').
- **autocommit**: Whether or not to commit outstanding transactions at the end of the request cycle (default: True).
- **dictrows**: Whether or not to support dict-like access to row objects (default: True).

You can override each of these values on a per-route basis:

```
@app.route('/cache/:item', sqlite={'dbfile': ':memory:'})
def cache(item, db):
    ...
```

or install two plugins with different keyword settings to the same application:

```

app = bottle.Bottle()
test_db = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
cache_db = bottle.ext.sqlite.Plugin(dbfile=':memory:', keyword='cache')
app.install(test_db)
app.install(cache_db)

@app.route('/show/:item')
def show(item, db):
    ...

@app.route('/cache/:item')
def cache(item, cache):
    ...

```

### 1.4.2 Bottle-Werkzeug

**Werkzeug** is a powerfull WSGI utility library for Python. It includes an interactive debugger and feature-packed request and response objects.

This plugin integrates `werkzeug.wrappers.Request` and `werkzeug.wrappers.Response` as an alternative to the built-in implementations, adds support for `werkzeug.exceptions` and replaces the default error page with an interactive debugger.

#### Installation

Install with one of the following commands:

```

$ pip install bottle-werkzeug
$ easy_install bottle-werkzeug

```

or download the latest version from github:

```

$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/werkzeug
$ python setup.py install

```

#### Usage

Once installed to an application, this plugin adds support for `werkzeug.wrappers.Response`, all kinds of `werkzeug.exceptions` and provides a thread-local instance of `werkzeug.wrappers.Request` that is updated with each request. The plugin instance itself doubles as a `werkzeug` module object, so you don't have to import `werkzeug` in your application. Here is an example:

```

import bottle

app = bottle.Bottle()
werkzeug = bottle.ext.werkzeug.Plugin()
app.install(werkzeug)

req = werkzeug.request # For the lazy.

@app.route('/hello/:name')
def say_hello(name):
    greet = {'en': 'Hello', 'de': 'Hallo', 'fr': 'Bonjour'}
    language = req.accept_languages.best_match(greet.keys())

```

```
if language:
    return werkzeug.Response('%s %s!' % (greet[language], name))
else:
    raise werkzeug.exceptions.NotAcceptable()
```

## Using the Debugger

This plugin replaces the default error page with an advanced debugger. If you have the *evalex* feature enabled, you will get an interactive console that allows you to inspect the error context in the browser. Please read Debugging Applications with werkzeug before you enable this feature.

## Configuration

The following configuration options exist for the plugin class:

- **evalex**: Enable the exception evaluation feature (interactive debugging). This requires a non-forking server and is a security risk. Please read Debugging Applications with werkzeug. (default: False)
- **request\_class**: Defaults to `werkzeug.wrappers.Request`
- **debugger\_class**: Defaults to a subclass of `werkzeug.debug.DebuggedApplication` which obeys the `bottle.DEBUG` setting.

---

## Knowledge Base

---

A collection of articles, guides and HOWTOs.

### 2.1 Tutorial: Todo-List Application

---

**Note:** This tutorial is a work in progress and written by [noisefloor](#).

---

This tutorial should give a brief introduction to the [Bottle](#) WSGI Framework. The main goal is to be able, after reading through this tutorial, to create a project using Bottle. Within this document, not all abilities will be shown, but at least the main and important ones like routing, utilizing the Bottle template abilities to format output and handling GET / POST parameters.

To understand the content here, it is not necessary to have a basic knowledge of WSGI, as Bottle tries to keep WSGI away from the user anyway. You should have a fair understanding of the [Python](#) programming language. Furthermore, the example used in the tutorial retrieves and stores data in a [SQL](#) database, so a basic idea about SQL helps, but is not a must to understand the concepts of Bottle. Right here, [SQLite](#) is used. The output of Bottle sent to the browser is formatted in some examples by the help of HTML. Thus, a basic idea about the common HTML tags does help as well.

For the sake of introducing Bottle, the Python code “in between” is kept short, in order to keep the focus. Also all code within the tutorial is working fine, but you may not necessarily use it “in the wild”, e.g. on a public web server. In order to do so, you may add e.g. more error handling, protect the database with a password, test and escape the input etc.

#### Table of Contents

- *Tutorial: Todo-List Application*
  - *Goals*
  - *Before We Start...*
  - *Using Bottle for a Web-Based ToDo List*
  - *Server Setup*
  - *Final Words*
  - *Complete Example Listing*

### 2.1.1 Goals

At the end of this tutorial, we will have a simple, web-based ToDo list. The list contains a text (with max 100 characters) and a status (0 for closed, 1 for open) for each item. Through the web-based user interface, open items can be view and edited and new items can be added.

During development, all pages will be available on `localhost` only, but later on it will be shown how to adapt the application for a “real” server, including how to use with Apache’s `mod_wsgi`.

Bottle will do the routing and format the output, with the help of templates. The items of the list will be stored inside a SQLite database. Reading and writing the database will be done by Python code.

We will end up with an application with the following pages and functionality:

- start page `http://localhost:8080/todo`
- adding new items to the list: `http://localhost:8080/new`
- page for editing items: `http://localhost:8080/edit/:no`
- validating data assigned by dynamic routes with the `@validate` decorator
- catching errors

### 2.1.2 Before We Start...

#### Install Bottle

Assuming that you have a fairly new installation of Python (version 2.5 or higher), you only need to install Bottle in addition to that. Bottle has no other dependencies than Python itself.

You can either manually install Bottle or use Python’s `easy_install`: `easy_install bottle`

#### Further Software Necessities

As we use SQLite3 as a database, make sure it is installed. On Linux systems, most distributions have SQLite3 installed by default. SQLite is available for Windows and MacOS X as well and the `sqlite3` module is part of the python standard library.

#### Create An SQL Database

First, we need to create the database we use later on. To do so, save the following script in your project directory and run it with python. You can use the interactive interpreter too:

```
import sqlite3
con = sqlite3.connect('todo.db') # Warning: This file is created in the current directory
con.execute("CREATE TABLE todo (id INTEGER PRIMARY KEY, task char(100) NOT NULL, status bool NOT NULL)")
con.execute("INSERT INTO todo (task,status) VALUES ('Read A-byte-of-python to get a good introduction to Python')")
con.execute("INSERT INTO todo (task,status) VALUES ('Visit the Python website',1)")
con.execute("INSERT INTO todo (task,status) VALUES ('Test various editors for and check the syntax highlighter')")
con.execute("INSERT INTO todo (task,status) VALUES ('Choose your favorite WSGI-Framework',0)")
```

This generates a database-file `todo.db` with tables called `todo` and three columns `id`, `task`, and `status`. `id` is a unique id for each row, which is used later on to reference the rows. The column `task` holds the text which describes the task, it can be max 100 characters long. Finally, the column `status` is used to mark a task as open (value 1) or closed (value 0).



### 2.1.3 Using Bottle for a Web-Based ToDo List

Now it is time to introduce Bottle in order to create a web-based application. But first, we need to look into a basic concept of Bottle: routes.

#### Understanding routes

Basically, each page visible in the browser is dynamically generated when the page address is called. Thus, there is no static content. That is exactly what is called a “route” within Bottle: a certain address on the server. So, for example, when the page `http://localhost:8080/todo` is called from the browser, Bottle “grabs” the call and checks if there is any (Python) function defined for the route “todo”. If so, Bottle will execute the corresponding Python code and return its result.

#### First Step - Showing All Open Items

So, after understanding the concept of routes, let’s create the first one. The goal is to see all open items from the ToDo list:

```
import sqlite3
from bottle import route, run

@route('/todo')
def todo_list():
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT id, task FROM todo WHERE status LIKE '1'")
    result = c.fetchall()
    return str(result)

run()
```

Save the code a `todo.py`, preferably in the same directory as the file `todo.db`. Otherwise, you need to add the path to `todo.db` in the `sqlite3.connect()` statement.

Let’s have a look what we just did: We imported the necessary module `sqlite3` to access to SQLite database and from Bottle we imported `route` and `run`. The `run()` statement simply starts the web server included in Bottle. By default, the web server serves the pages on localhost and port 8080. Furthermore, we imported `route`, which is the function responsible for Bottle’s routing. As you can see, we defined one function, `todo_list()`, with a few lines of code reading from the database. The important point is the **decorator statement** `@route('/todo')` right before the `def todo_list()` statement. By doing this, we bind this function to the route `/todo`, so every time the browsers calls `http://localhost:8080/todo`, Bottle returns the result of the function `todo_list()`. That is how routing within bottle works.

Actually you can bind more than one route to a function. So the following code:

```
@route('/todo')
@route('/my_todo_list')
def todo_list():
    ...
```

will work fine, too. What will not work is to bind one route to more than one function.

What you will see in the browser is what is returned, thus the value given by the `return` statement. In this example, we need to convert `result` in to a string by `str()`, as Bottle expects a string or a list of strings from the return statement. But here, the result of the database query is a list of tuples, which is the standard defined by the [Python DB API](#).

Now, after understanding the little script above, it is time to execute it and watch the result yourself. Remember that on Linux- / Unix-based systems the file `todo.py` needs to be executable first. Then, just run `python todo.py` and call the page `http://localhost:8080/todo` in your browser. In case you made no mistake writing the script, the output should look like this:

```
[(2, u'Visit the Python website'), (3, u'Test various editors for and check the syntax highlighting')]
```

If so - congratulations! You are now a successful user of Bottle. In case it did not work and you need to make some changes to the script, remember to stop Bottle serving the page, otherwise the revised version will not be loaded.

Actually, the output is not really exciting nor nice to read. It is the raw result returned from the SQL query.

So, in the next step we format the output in a nicer way. But before we do that, we make our life easier.

### Debugging and Auto-Reload

Maybe you already noticed that Bottle sends a short error message to the browser in case something within the script is wrong, e.g. the connection to the database is not working. For debugging purposes it is quite helpful to get more details. This can be easily achieved by adding the following statement to the script:

```
from bottle import run, route, debug
...
#add this at the very end:
debug(True)
run()
```

By enabling “debug”, you will get a full stacktrace of the Python interpreter, which usually contains useful information for finding bugs. Furthermore, templates (see below) are not cached, thus changes to templates will take effect without stopping the server.

**Warning:** That `debug(True)` is supposed to be used for development only, it should *not* be used in production environments.

A further quiet nice feature is auto-reloading, which is enabled by modifying the `run()` statement to

```
run(reloader=True)
```

This will automatically detect changes to the script and reload the new version once it is called again, without the need to stop and start the server.

Again, the feature is mainly supposed to be used while development, not on productive systems.

### Bottle Template To Format The Output

Now let’s have a look at casting the output of the script into a proper format.

Actually Bottle expects to receive a string or a list of strings from a function and returns them by the help of the built-in server to the browser. Bottle does not bother about the content of the string itself, so it can be text formatted with HTML markup, too.

Bottle brings its own easy-to-use template engine with it. Templates are stored as separate files having a `.tpl` extension. The template can be called then from within a function. Templates can contain any type of text (which will be most likely HTML-markup mixed with Python statements). Furthermore, templates can take arguments, e.g. the result set of a database query, which will be then formatted nicely within the template.

Right here, we are going to cast the result of our query showing the open ToDo items into a simple table with two columns: the first column will contain the ID of the item, the second column the text. The result set is, as seen above, a list of tuples, each tuple contains one set of results.

To include the template in our example, just add the following lines:

```
from bottle import route, run, debug, template
...
result = c.fetchall()
c.close()
output = template('make_table', rows=result)
return output
...
```

So we do here two things: first, we import `template` from Bottle in order to be able to use templates. Second, we assign the output of the template `make_table` to the variable `output`, which is then returned. In addition to calling the template, we assign `result`, which we received from the database query, to the variable `rows`, which is later on used within the template. If necessary, you can assign more than one variable / value to a template.

Templates always return a list of strings, thus there is no need to convert anything. Of course, we can save one line of code by writing `return template('make_table', rows=result)`, which gives exactly the same result as above.

Now it is time to write the corresponding template, which looks like this:

```

%#template to generate a HTML table from a list of tuples (or list of lists, or tuple of tuples or .
<p>The open items are as follows:</p>
<table border="1">
%for row in rows:
    <tr>
        %for r in row:
            <td>{{r}}</td>
        %end
    </tr>
%end
</table>

```

Save the code as `make_table.tpl` in the same directory where `todo.py` is stored.

Let's have a look at the code: every line starting with `%` is interpreted as Python code. Please note that, of course, only valid Python statements are allowed, otherwise the template will raise an exception, just as any other Python code. The other lines are plain HTML markup.

As you can see, we use Python's `for` statement two times, in order to go through `rows`. As seen above, `rows` is a variable which holds the result of the database query, so it is a list of tuples. The first `for` statement accesses the tuples within the list, the second one the items within the tuple, which are put each into a cell of the table. It is important that you close all `for`, `if`, `while` etc. statements with `%end`, otherwise the output may not be what you expect.

If you need to access a variable within a non-Python code line inside the template, you need to put it into double curly braces. This tells the template to insert the actual value of the variable right in place.

Run the script again and look at the output. Still not really nice, but at least more readable than the list of tuples. Of course, you can spice-up the very simple HTML markup above, e.g. by using in-line styles to get a better looking output.

## Using GET and POST Values

As we can review all open items properly, we move to the next step, which is adding new items to the ToDo list. The new item should be received from a regular HTML-based form, which sends its data by the GET method.

To do so, we first add a new route to our script and tell the route that it should get GET data:

```
from bottle import route, run, debug, template, request
...
return template('make_table', rows=result)
...

@route('/new', method='GET')
def new_item():

    new = request.GET.get('task', '').strip()

    conn = sqlite3.connect('todo.db')
    c = conn.cursor()

    c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
    new_id = c.lastrowid

    conn.commit()
    c.close()

    return '<p>The new task was inserted into the database, the ID is %s</p>' % new_id
```

To access GET (or POST) data, we need to import `request` from Bottle. To assign the actual data to a variable, we use the statement `request.GET.get('task', '').strip()` statement, where `task` is the name of the GET data we want to access. That's all. If your GET data has more than one variable, multiple `request.GET.get()` statements can be used and assigned to other variables.

The rest of this piece of code is just processing of the gained data: writing to the database, retrieve the corresponding id from the database and generate the output.

But where do we get the GET data from? Well, we can use a static HTML page holding the form. Or, what we do right now, is to use a template which is output when the route `/new` is called without GET data.

The code needs to be extended to:

```
...
@route('/new', method='GET')
def new_item():

    if request.GET.get('save', '').strip():

        new = request.GET.get('task', '').strip()
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()

        c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
        new_id = c.lastrowid

        conn.commit()
        c.close()

        return '<p>The new task was inserted into the database, the ID is %s</p>' % new_id
    else:
        return template('new_task.tpl')
```

`new_task.tpl` looks like this:

```
<p>Add a new task to the ToDo list:</p>
<form action="/new" method="GET">
<input type="text" size="100" maxlength="100" name="task">
```

```
<input type="submit" name="save" value="save">
</form>
```

That's all. As you can see, the template is plain HTML this time.

Now we are able to extend our to do list.

By the way, if you prefer to use POST data: this works exactly the same way, just use `request.POST.get()` instead.

## Editing Existing Items

The last point to do is to enable editing of existing items.

By using only the routes we know so far it is possible, but may be quite tricky. But Bottle knows something called “dynamic routes”, which makes this task quite easy.

The basic statement for a dynamic route looks like this:

```
@route('/myroute/:something')
```

The key point here is the colon. This tells Bottle to accept for `:something` any string up to the next slash. Furthermore, the value of `something` will be passed to the function assigned to that route, so the data can be processed within the function.

For our ToDo list, we will create a route `@route('/edit/:no')`, where `no` is the id of the item to edit.

The code looks like this:

```
@route('/edit/:no', method='GET')
def edit_item(no):

    if request.GET.get('save', '').strip():
        edit = request.GET.get('task', '').strip()
        status = request.GET.get('status', '').strip()

        if status == 'open':
            status = 1
        else:
            status = 0

        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("UPDATE todo SET task = ?, status = ? WHERE id LIKE ?", (edit, status, no))
        conn.commit()

        return '<p>The item number %s was successfully updated</p>' % no
    else:
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (str(no)))
        cur_data = c.fetchone()

        return template('edit_task', old=cur_data, no=no)
```

It is basically pretty much the same what we already did above when adding new items, like using GET data etc. The main addition here is using the dynamic route `:no`, which here passes the number to the corresponding function. As you can see, `no` is used within the function to access the right row of data within the database.

The template `edit_task.tpl` called within the function looks like this:

```
##template for editing a task
##the template expects to receive a value for "no" as well a "old", the text of the selected ToDo item
<p>Edit the task with ID = {{no}}</p>
<form action="/edit/{{no}}" method="get">
<input type="text" name="task" value="{{old[0]}}" size="100" maxlength="100">
<select name="status">
<option>open</option>
<option>closed</option>
</select>
<br/>
<input type="submit" name="save" value="save">
</form>
```

Again, this template is a mix of Python statements and HTML, as already explained above.

A last word on dynamic routes: you can even use a regular expression for a dynamic route. But this topic is not discussed further here.

### Validating Dynamic Routes

Using dynamic routes is fine, but for many cases it makes sense to validate the dynamic part of the route. For example, we expect an integer number in our route for editing above. But if a float, characters or so are received, the Python interpreter throws an exception, which is not what we want.

For those cases, Bottle offers the `@validate` decorator, which validates the “input” prior to passing it to the function. In order to apply the validator, extend the code as follows:

```
from bottle import route, run, debug, template, request, validate
...
@route('/edit/:no', method='GET')
@validate(no=int)
def edit_item(no):
...
```

At first, we imported `validate` from the Bottle framework, than we apply the `@validate`-decorator. Right here, we validate if `no` is an integer. Basically, the validation works with all types of data like floats, lists etc.

Save the code and call the page again using a “403 forbidden” value for `:no`, e.g. a float. You will receive not an exception, but a “403 - Forbidden” error, saying that an integer was expected.

### Dynamic Routes Using Regular Expressions

Bottle can also handle dynamic routes, where the “dynamic part” of the route can be a regular expression.

So, just to demonstrate that, let’s assume that all single items in our ToDo list should be accessible by their plain number, by a term like e.g. “item1”. For obvious reasons, you do not want to create a route for every item. Furthermore, the simple dynamic routes do not work either, as part of the route, the term “item” is static.

As said above, the solution is a regular expression:

```
@route('/item:item#[1-9]+#')
def show_item(item):
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (item))
    result = c.fetchall()
    c.close()
```

```

if not result:
    return 'This item number does not exist!'
else:
    return 'Task: %s' %result[0]

```

Of course, this example is somehow artificially constructed - it would be easier to use a plain dynamic route only combined with a validation. Nevertheless, we want to see how regular expression routes work: the line `@route('/item:item_[1-9]+#')` starts like a normal route, but the part surrounded by `#` is interpreted as a regular expression, which is the dynamic part of the route. So in this case, we want to match any digit between 0 and 9. The following function “`show_item`” just checks whether the given item is present in the database or not. In case it is present, the corresponding text of the task is returned. As you can see, only the regular expression part of the route is passed forward. Furthermore, it is always forwarded as a string, even if it is a plain integer number, like in this case.

## Returning Static Files

Sometimes it may become necessary to associate a route not to a Python function, but just return a static file. So if you have for example a help page for your application, you may want to return this page as plain HTML. This works as follows:

```

from bottle import route, run, debug, template, request, validate, send_file

@route('/help')
def help():
    send_file('help.html', root='/path/to/file')

```

At first, we need to import `send_file` from Bottle. As you can see, the `send_file` statement replaces the `return` statement. It takes at least two arguments: the name of the file to be returned and the path to the file. Even if the file is in the same directory as your application, the path needs to be stated. But in this case, you can use `'.'` as a path, too. Bottle guesses the MIME-type of the file automatically, but in case you like to state it explicitly, add a third argument to `send_file`, which would be here `mimetype='text/html'`. `send_file` works with any type of route, including the dynamic ones.

## Returning JSON Data

There may be cases where you do not want your application to generate the output directly, but return data to be processed further on, e.g. by JavaScript. For those cases, Bottle offers the possibility to return JSON objects, which is sort of standard for exchanging data between web applications. Furthermore, JSON can be processed by many programming languages, including Python

So, let's assume we want to return the data generated in the regular expression route example as a JSON object. The code looks like this:

```

@route('/json:json_[1-9]+#')
def show_json(json):
    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (item))
    result = c.fetchall()
    c.close()

    if not result:
        return {'task': 'This item number does not exist!'}
    else:
        return {'Task': result[0]}

```

As you can, that is fairly simple: just return a regular Python dictionary and Bottle will convert it automatically into a JSON object prior to sending. So if you e.g. call “<http://localhost/json1>” Bottle should in this case return the JSON object `{"Task": ["Read A-byte-of-python to get a good introduction into Python"]}`.

## Catching Errors

The next step may is to catch the error with Bottle itself, to keep away any type of error message from the user of your application. To do that, Bottle has an “error-route”, which can be assigned to a HTML-error.

In our case, we want to catch a 403 error. The code is as follows:

```
from bottle import error

@error(403)
def mistake(code):
    return 'The parameter you passed has the wrong format!'
```

So, at first we need to import `error` from Bottle and define a route by `error(403)`, which catches all “403 forbidden” errors. The function “mistake” is assigned to that. Please note that `error()` always passes the error-code to the function - even if you do not need it. Thus, the function always needs to accept one argument, otherwise it will not work.

Again, you can assign more than one error-route to a function, or catch various errors with one function each. So this code:

```
@error(404)
@error(403)
def mistake(code):
    return 'There is something wrong!'
```

works fine, the following one as well:

```
@error(403)
def mistake403(code):
    return 'The parameter you passed has the wrong format!'

@error(404)
def mistake404(code):
    return 'Sorry, this page does not exist!'
```

## Summary

After going through all the sections above, you should have a brief understanding how the Bottle WSGI framework works. Furthermore you have all the knowledge necessary to use Bottle for your applications.

The following chapter give a short introduction how to adapt Bottle for larger projects. Furthermore, we will show how to operate Bottle with web servers which perform better on a higher load / more web traffic than the one we used so far.

## 2.1.4 Server Setup

So far, we used the standard server used by Bottle, which is the [WSGI reference Server](#) shipped along with Python. Although this server is perfectly suitable for development purposes, it is not really suitable for larger applications. But before we have a look at the alternatives, let’s have a look how to tweak the setting of the standard server first



## Running Bottle on a different port and IP

As standard, Bottle serve the pages on the IP adress 127.0.0.1, also known as `localhost`, and on port 8080. To modify the setting is pretty simple, as additional parameters can be passed to Bottle's `run()` function to change the port and the address.

To change the port, just add `port=portnumber` to the run command. So, for example:

```
run(port=80)
```

would make Bottle listen to port 80.

To change the IP address where Bottle is listening:

```
run(host='123.45.67.89')
```

Of course, both parameters can be combined, like:

```
run(port=80, host='123.45.67.89')
```

The `port` and `host` parameter can also be applied when Bottle is running with a different server, as shown in the following section.

## Running Bottle with a different server

As said above, the standard server is perfectly suitable for development, personal use or a small group of people only using your application based on Bottle. For larger tasks, the standard server may become a bottleneck, as it is single-threaded, thus it can only serve one request at a time.

But Bottle has already various adapters to multi-threaded servers on board, which perform better on higher load. Bottle supports [CherryPy](#), [Fapws3](#), [Flup](#) and [Paste](#).

If you want to run for example Bottle with the past server, use the following code:

```
from bottle import PasteServer
...
run(server=PasteServer)
```

This works exactly the same way with `FlupServer`, `CherryPyServer` and `FapwsServer`.

## Running Bottle on Apache with mod\_wsgi

Maybe you already have an [Apache](#) or you want to run a Bottle-based application large scale - then it is time to think about Apache with `mod_wsgi`.

We assume that your Apache server is up and running and `mod_wsgi` is working fine as well. On a lot of Linux distributions, `mod_wsgi` can be installed via the package management easily.

Bottle brings an adapter for `mod_wsgi` with it, so serving your application is an easy task.

In the following example, we assume that you want to make your application “ToDo list” accessible through `http://www.mypage.com/todo` and your code, templates and SQLite database are stored in the path `/var/www/todo`.

When you run your application via `mod_wsgi`, it is imperative to remove the `run()` statement from your code, otherwise it won't work here.

After that, create a file called `adapter.wsgi` with the following content:

```
import sys, os, bottle

sys.path = ['/var/www/todo/'] + sys.path
os.chdir(os.path.dirname(__file__))

import todo # This loads your application

application = bottle.default_app()
```

and save it in the same path, `/var/www/todo`. Actually the name of the file can be anything, as long as the extension is `.wsgi`. The name is only used to reference the file from your virtual host.

Finally, we need to add a virtual host to the Apache configuration, which looks like this:

```
<VirtualHost *>
    ServerName mypage.com

    WSGIDaemonProcess todo user=www-data group=www-data processes=1 threads=5
    WSGIScriptAlias / /var/www/todo/adapter.wsgi

    <Directory /var/www/todo>
        WSGIProcessGroup todo
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

After restarting the server, your ToDo list should be accessible at `http://www.mypage.com/todo`

## 2.1.5 Final Words

Now we are at the end of this introduction and tutorial to Bottle. We learned about the basic concepts of Bottle and wrote a first application using the Bottle framework. In addition to that, we saw how to adapt Bottle for large task and servr Bottle through an Apache web server with `mod_wsgi`.

As said in the introduction, this tutorial is not showing all shades and possibilities of Bottle. What we skipped here is e.g. receiving file objects and streams and how to handle authentication data. Furthermore, we did not show how templates can be called from within another template. For an introduction into those points, please refer to the full [Bottle documentation](#).

## 2.1.6 Complete Example Listing

As the ToDo list example was developed piece by piece, here is the complete listing:

Main code for the application `todo.py`:

```
import sqlite3
from bottle import route, run, debug, template, request, validate, send_file, error

# only needed when you run Bottle on mod_wsgi
from bottle import default_app

@route('/todo')
def todo_list():

    conn = sqlite3.connect('todo.db')
```

```

c = conn.cursor()
c.execute("SELECT id, task FROM todo WHERE status LIKE '1';")
result = c.fetchall()
c.close()

output = template('make_table', rows=result)
return output

@route('/new', method='GET')
def new_item():

    if request.GET.get('save', '').strip():

        new = request.GET.get('task', '').strip()
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()

        c.execute("INSERT INTO todo (task,status) VALUES (?,?)", (new,1))
        new_id = c.lastrowid

        conn.commit()
        c.close()

        return '<p>The new task was inserted into the database, the ID is %s</p>' % new_id

    else:
        return template('new_task.tpl')

@route('/edit/:no', method='GET')
@validate(no=int)
def edit_item(no):

    if request.GET.get('save', '').strip():
        edit = request.GET.get('task', '').strip()
        status = request.GET.get('status', '').strip()

        if status == 'open':
            status = 1
        else:
            status = 0

        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("UPDATE todo SET task = ?, status = ? WHERE id LIKE ?", (edit,status,no))
        conn.commit()

        return '<p>The item number %s was successfully updated</p>' % no

    else:
        conn = sqlite3.connect('todo.db')
        c = conn.cursor()
        c.execute("SELECT task FROM todo WHERE id LIKE ?", (str(no)))
        cur_data = c.fetchone()

        return template('edit_task', old = cur_data, no = no)

@route('/item:item#[1-9]+#')
def show_item(item):

```

```
conn = sqlite3.connect('todo.db')
c = conn.cursor()
c.execute("SELECT task FROM todo WHERE id LIKE ?", (item))
result = c.fetchall()
c.close()

if not result:
    return 'This item number does not exist!'
else:
    return 'Task: %s' %result[0]

@route('/help')
def help():

    send_file('help.html', root='.')

@route('/json:json#[1-9]+#')
def show_json(json):

    conn = sqlite3.connect('todo.db')
    c = conn.cursor()
    c.execute("SELECT task FROM todo WHERE id LIKE ?", (json))
    result = c.fetchall()
    c.close()

    if not result:
        return {'task': 'This item number does not exist!'}
    else:
        return {'Task': result[0]}

@error(403)
def mistake403(code):
    return 'There is a mistake in your url!'

@error(404)
def mistake404(code):
    return 'Sorry, this page does not exist!'

debug(True)
run(reloader=True)
#remember to remove reloader=True and debug(True) when you move your application from development to
```

Template make\_table.tpl:

```
%#template to generate a HTML table from a list of tuples (or list of lists, or tuple of tuples or .
<p>The open items are as follows:</p>
<table border="1">
%for row in rows:
    <tr>
    %for r in row:
        <td>{{r}}</td>
    %end
    </tr>
%end
</table>
```

Template `edit_task.tpl`:

```
##template for editing a task
##the template expects to receive a value for "no" as well a "old", the text of the selected ToDo item
<p>Edit the task with ID = {{no}}</p>
<form action="/edit/{{no}}" method="get">
<input type="text" name="task" value="{{old[0]}}" size="100" maxlength="100">
<select name="status">
<option>open</option>
<option>closed</option>
</select>
<br/>
<input type="submit" name="save" value="save">
</form>
```

Template `new_task.tpl`:

```
##template for the form for a new task
<p>Add a new task to the ToDo list:</p>
<form action="/new" method="GET">
<input type="text" size="100" maxlenght="100" name="task">
<input type="submit" name="save" value="save">
</form>
```

## 2.2 Primer to Asynchronous Applications

Asynchronous design patterns don't mix well with the synchronous nature of [WSGI](#). This is why most asynchronous frameworks (tornado, twisted, ...) implement a specialized API to expose their asynchronous features. Bottle is a WSGI framework and shares the synchronous nature of WSGI, but thanks to the awesome [gevent project](#), it is still possible to write asynchronous applications with bottle. This article documents the usage of Bottle with Asynchronous WSGI.

### 2.2.1 The Limits of Synchronous WSGI

Briefly worded, the [WSGI specification \(pep 3333\)](#) defines a request/response circle as follows: The application callable is invoked once for each request and must return a body iterator. The server then iterates over the body and writes each chunk to the socket. As soon as the body iterator is exhausted, the client connection is closed.

Simple enough, but there is a snag: All this happens synchronously. If your application needs to wait for data (IO, sockets, databases, ...), it must either yield empty strings (busy wait) or block the current thread. Both solutions occupy the handling thread and prevent it from answering new requests. There is consequently only one ongoing request per thread.

Most servers limit the number of threads to avoid their relatively high overhead. Pools of 20 or less threads are common. As soon as all threads are occupied, any new connection is stalled. The server is effectively dead for everyone else. If you want to implement a chat that uses long-polling ajax requests to get real-time updates, you'd reach the limited at 20 concurrent connections. That's a pretty small chat.

### 2.2.2 Greenlets for the rescue

Most servers limit the size of their worker pools to a relatively low number of concurrent threads, due to the high overhead involved in switching between and creating new threads. While threads are cheap compared to processes (forks), they are still expensive to create for each new connection.

The `gevent` module adds *greenlets* to the mix. Greenlets behave similar to traditional threads, but are very cheap to create. A gevent-based server can spawn thousands of greenlets (one for each connection) with almost no overhead. Blocking individual greenlets has no impact on the servers ability to accept new requests. The number of concurrent connections is virtually unlimited.

This makes creating asynchronous applications incredibly easy, because they look and feel like synchronous applications. A gevent-based server is actually not asynchronous, but massively multi-threaded. Here is an example:

```
from gevent import monkey; monkey.patch_all()

from time import sleep
from bottle import route, run

@route('/stream')
def stream():
    yield 'START'
    sleep(3)
    yield 'MIDDLE'
    sleep(5)
    yield 'END'

run(host='0.0.0.0', port=8080, server='gevent')
```

The first line is important. It causes gevent to monkey-patch most of Python's blocking APIs to not block the current thread, but pass the CPU to the next greenlet instead. It actually replaces Python's threading with gevent-based pseudo-threads. This is why you can still use `time.sleep()` which would normally block the whole thread. If you don't feel comfortable with monkey-patching python build-ins, you can use the corresponding gevent functions (`gevent.sleep()` in this case).

If you run this script and point your browser to `http://localhost:8080/stream`, you should see *START*, *MIDDLE*, and *END* show up one by one (rather than waiting 8 seconds to see them all at once). It works exactly as with normal threads, but now your server can handle thousands of concurrent requests without any problems.

---

**Note:** Some browsers buffer a certain amount of data before they start rendering a page. You might need to yield more than a few bytes to see an effect in these browsers. Additionally, many browsers have a limit of one concurrent connection per URL. If this is the case, you can use a second browser or a benchmark tool (e.g. *ab* or *httpperf*) to measure performance.

---

## 2.2.3 Event Callbacks

A very common design pattern in asynchronous frameworks (including tornado, twisted, node.js and friends) is to use non-blocking APIs and bind callbacks to asynchronous events. The socket object is kept open until it is closed explicitly to allow callbacks to write to the socket at a later point. Here is an example based on the [tornado library](#):

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        worker = SomeAsyncWorker()
        worker.on_data(lambda chunk: self.write(chunk))
        worker.on_finish(lambda: self.finish())
```

The main benefit is that the request handler terminates early. The handling thread can move on and accept new requests while the callbacks continue to write to sockets of previous requests. This is how these frameworks manage to process a lot of concurrent requests with only a small number of OS threads.

With Gevent+WSGI, things are different: First, terminating early has no benefit because we have an unlimited pool of (pseudo)threads to accept new connections. Second, we cannot terminate early because that would close the socket

(as required by WSGI). Third, we must return an iterable to conform to WSGI.

In order to conform to the WSGI standard, all we have to do is to return a body iterable that we can write to asynchronously. With the help of `gevent.queue`, we can *simulate* a detached socket and rewrite the previous example as follows:

```
@route('/fetch')
def fetch():
    body = gevent.queue.Queue()
    worker = SomeAsyncWorker()
    worker.on_data(lambda chunk: body.put(chunk))
    worker.on_finish(lambda: body.put(StopIteration))
    return body
```

From the server perspective, the queue object is iterable, blocks if empty and stops as soon as it reaches `StopIteration`. This conforms to WSGI. On application side, the queue object behaves like a non-blocking socket. You can write to it at any time, pass it around and even start a new (pseudo)thread that writes to it asynchronously. This is how long-polling is implemented most of the time.

If the demand is high enough, I could port the [gevent long-polling chat example](#) to bottle. Join the [mailing-list](#) if you have questions or want to help.

## 2.3 Recipes

This is a collection of code snippets and examples for common use cases.

### 2.3.1 Keeping track of Sessions

There is no built-in support for sessions because there is no *right* way to do it (in a micro framework). Depending on requirements and environment you could use [beaker](#) middleware with a fitting backend or implement it yourself. Here is an example for beaker sessions with a file-based backend:

```
import bottle
from beaker.middleware import SessionMiddleware

session_opts = {
    'session.type': 'file',
    'session.cookie_expires': 300,
    'session.data_dir': './data',
    'session.auto': True
}

app = SessionMiddleware(bottle.app(), session_opts)

@bottle.route('/test')
def test():
    s = bottle.request.environ.get('beaker.session')
    s['test'] = s.get('test', 0) + 1
    s.save()
    return 'Test counter: %d' % s['test']

bottle.run(app=app)
```

### 2.3.2 Debugging with Style: Debugging Middleware

Bottle catches all Exceptions raised in your app code to prevent your WSGI server from crashing. If the built-in `debug()` mode is not enough and you need exceptions to propagate to a debugging middleware, you can turn off this behaviour:

```
import bottle
app = bottle.app()
app.catchall = False #Now most exceptions are re-raised within bottle.
myapp = DebuggingMiddleware(app) #Replace this with a middleware of your choice (see below)
bottle.run(app=myapp)
```

Now, bottle only catches its own exceptions (`HTTPError`, `HTTPResponse` and `BottleException`) and your middleware can handle the rest.

The `werkzeug` and `paste` libraries both ship with very powerfull debugging WSGI middleware. Look at `werkzeug.debug.DebuggedApplication` for `werkzeug` and `paste.evalexception.middleware.EvalException` for `paste`. They both allow you do inspect the stack and even execute python code within the stack context, so **do not use them in production**.

### 2.3.3 Embedding other WSGI Apps

This is not the recommend way (you should use a middleware in front of bottle to do this) but you can call other WSGI applications from within your bottle app and let bottle act as a pseudo-middleware. Here is an example:

```
from bottle import request, response, route
subproject = SomeWSGIApplication()

@route('/subproject/:subpath#.*#', method='ALL')
def call_wsgi(subpath):
    new_environ = request.environ.copy()
    new_environ['SCRIPT_NAME'] = new_environ.get('SCRIPT_NAME', '') + '/subproject'
    new_environ['PATH_INFO'] = '/' + subpath
    def start_response(status, headerlist):
        response.status = int(status.split()[0])
        for key, value in headerlist:
            response.add_header(key, value)
    return app(new_environ, start_response)
```

Again, this is not the recommend way to implement subprojects. It is only here because many people asked for this and to show how bottle maps to WSGI.

### 2.3.4 Ignore trailing slashes

For Bottle, `/example` and `/example/` are two different routes <sup>1</sup>. To treat both URLs the same you can add two `@route` decorators:

```
@route('/test')
@route('/test/')
def test(): return 'Slash? no?'
```

or add a WSGI middleware that strips trailing slashes from all URLs:

---

<sup>1</sup> Because they are. See <<http://www.ietf.org/rfc/rfc3986.txt>>



```
class StripPathMiddleware(object):
    def __init__(self, app):
        self.app = app
    def __call__(self, e, h):
        e['PATH_INFO'] = e['PATH_INFO'].rstrip('/')
        return self.app(e, h)

app = bottle.app()
myapp = StripPathMiddleware(app)
bottle.run(app=appmy)
```

## 2.3.5 Keep-alive requests

**Note:** For a more detailed explanation, see [Primer to Asynchronous Applications](#).

Several “push” mechanisms like XHR multipart need the ability to write response data without closing the connection in conjunction with the response header “Connection: keep-alive”. WSGI does not easily lend itself to this behavior, but it is still possible to do so in Bottle by using the `gevent` async framework. Here is a sample that works with either the `gevent` HTTP server or the `paste` HTTP server (it may work with others, but I have not tried). Just change `server='gevent'` to `server='paste'` to use the `paste` server:

```
from gevent import monkey; monkey.patch_all()

import time
from bottle import route, run

@route('/stream')
def stream():
    yield 'START'
    time.sleep(3)
    yield 'MIDDLE'
    time.sleep(5)
    yield 'END'

run(host='0.0.0.0', port=8080, server='gevent')
```

If you browse to `http://localhost:8080/stream`, you should see ‘START’, ‘MIDDLE’, and ‘END’ show up one at a time (rather than waiting 8 seconds to see them all at once).

## 2.3.6 Gzip Compression in Bottle

**Note:** For a detailed discussion, see [compression](#)

A common feature request is for Bottle to support Gzip compression, which speeds up sites by compressing static resources (like CSS and JS files) during a request.

Supporting Gzip compression is not a straightforward proposition, due to a number of corner cases that crop up frequently. A proper Gzip implementation must:

- Compress on the fly and be fast doing so.
- Do not compress for browsers that don’t support it.
- Do not compress files that are compressed already (images, videos).

- Do not compress dynamic files.
- Support two differed compression algorithms (gzip and deflate).
- Cache compressed files that don't change often.
- De-validate the cache if one of the files changed anyway.
- Make sure the cache does not get to big.
- Do not cache small files because a disk seek would take longer than on-the-fly compression.

Because of these requirements, it is the recommendation of the Bottle project that Gzip compression is best handled by the WSGI server Bottle runs on top of. WSGI servers such as [cherry.py](#) provide a [GzipFilter](#) middleware that can be used to accomplish this.

## 2.4 Frequently Asked Questions

### 2.4.1 About Bottle

#### Is bottle suitable for complex applications?

Bottle is a *micro* framework designed for prototyping and building small web applications and services. It stays out of your way and allows you to get things done fast, but misses some advanced features and ready-to-use solutions found in other frameworks (MVC, ORM, form validation, scaffolding, XML-RPC). Although it *is* possible to add these features and build complex applications with Bottle, you should consider using a full-stack Web framework like [pylons](#) or [paste](#) instead.

### 2.4.2 Common Problems and Pitfalls

#### “Template Not Found” in `mod_wsgi/mod_python`

Bottle searches in `./` and `./views/` for templates. In a `mod_python` or `mod_wsgi` environment, the working directory (`./`) depends on your Apache settings. You should add an absolute path to the template search path:

```
bottle.TEMPLATE_PATH.insert(0, '/absolut/path/to/templates/')
```

so bottle searches the right paths.

#### Dynamic Routes and Slashes

In *dynamic route syntax*, a placeholder token (`:name`) matches everything up to the next slash. This equals to `[^/]+` in regular expression syntax. To accept slashes too, you have to add a custom regular pattern to the placeholder. An example: `/images/:filepath#.*#` would match `/images/icons/error.png` but `/images/:filename` won't.

---

## Development and Contribution

---

These chapters are intended for developers interested in the bottle development and release workflow.

### 3.1 Release Notes and Changelog

#### 3.1.1 Release 0.9

##### Whats new?

- A brand new plugin-API. See *Plugins* and *Plugin Development Guide* for details.
- The `route()` decorator got a lot of new features. See `Bottle.route()` for details.
- New server adapters for *gevent*, *meinheld* and *bjoern*.
- Support for SimpleTAL templates.
- Better runtime exception handling for mako templates in debug mode.
- Lots of documentation, fixes and small improvements.
- A new `Request.urlparts` property.

##### Performance improvements

- The Router now special-cases `wsgi.run_once` environments to speed up CGI.
- Reduced module load time by ~30% and optimized template parser. See *8ccb2d*, *f72a7c* and *b14b9a* for details.
- Support for “App Caching” on Google App Engine. See *af93ec*.
- Some of the rarely used or deprecated features are now plugins that avoid overhead if the feature is not used.

##### API changes

This release is mostly backward compatible, but some APIs are marked deprecated now and will be removed for the next release. Most noteworthy:

- The `static` route parameter is deprecated. You can escape wild-cards with a backslash.
- Type-based output filters are deprecated. They can easily be replaced with plugins.

## Thanks to

Thanks to all the people who found bugs, sent patches, spread the word, helped each other on the mailing-list and made this release possible. You are awesome :) Really, you are.

I hope the following (alphabetically sorted) list is complete. If you miss your name on that list (or want your name removed) please `tell` me.

Adam R. Smith, Alexey Borzenkov, ‘apheage’, ‘BillMa’, Brandon Gilmore, Branko Vukelic, Damien De-gois, David Buxton, Duane Johnson, Frank Murphy, Ian Davis, Itamar Nabriski, ‘iurisilvio’, Jeff Nichols, Jeremy Kelley, ‘joegester’, Jonas Haag, ‘Karl’, ‘Kraken’, Kyle Fritz, ‘m35’, ‘masklinn’, [reddit](#), Santiago Gala, Sean M. Collins, ‘Seth’, Sigurd Høgsbro, Stuart Rackham, Sun Ning, Tomás A. Schertel, Tristan Zajonc, ‘voltron’ and Wieland Hoffmann

## 3.1.2 Release 0.8

### API changes

These changes may break compatibility with previous versions.

- The built-in Key/Value database is not available anymore. It is marked deprecated since 0.6.4
- The Route syntax and behaviour changed.
  - Regular expressions must be encapsulated with `#`. In 0.6 all non-alphanumeric characters not present in the regular expression were allowed.
  - Regular expressions not part of a route wildcard are escaped automatically. You don’t have to escape dots or other regular control characters anymore. In 0.6 the whole URL was interpreted as a regular expression. You can use anonymous wildcards (`/index: #(\.html)?#`) to achieve a similar behaviour.
- The `BreakTheBottle` exception is gone. Use `HTTPResponse` instead.
- The `SimpleTemplate` engine escapes HTML special characters in `{{bad_html}}` expressions automatically. Use the new `{{!good_html}}` syntax to get old behaviour (no escaping).
- The `SimpleTemplate` engine returns unicode strings instead of lists of byte strings.
- `bottle.optimize()` and the automatic route optimization is obsolete.
- Some functions and attributes were renamed: `* Request._environ` is now `Request.environ` \* `Response.header` is now `Response.headers` \* `default_app()` is obsolete. Use `app()` instead.
- The default `redirect()` code changed from 307 to 303.
- Removed support for `@default`. Use `@error(404)` instead.

### New features

This is an incomplete list of new features and improved functionality.

- The `Request` object got new properties: `Request.body`, `Request.auth`, `Request.url`, `Request.header`, `Request.forms`, `Request.files`.
- The `Response.set_cookie()` and `Request.get_cookie()` methods are now able to encode and decode python objects. This is called a *secure cookie* because the encoded values are signed and protected from changes on client side. All pickle-able data structures are allowed.

- The new `Router` class drastically improves performance for setups with lots of dynamic routes and supports named routes (named route + dict = URL string).
- It is now possible (and recommended) to return `HTTPError` and `HTTPResponse` instances or other exception objects instead of raising them.
- The new function `static_file()` equals `send_file()` but returns a `HTTPResponse` or `HTTPError` instead of raising it. `send_file()` is deprecated.
- New `get()`, `post()`, `put()` and `delete()` decorators.
- The `SimpleTemplate` engine got full unicode support.
- Lots of non-critical bugfixes.

## 3.2 Developer Notes

This document is intended for developers and package maintainers interested in the bottle development and release workflow. If you want to contribute, you are just right!

### 3.2.1 Get involved

There are several ways to join the community and stay up to date. Here are some of them:

- **Mailinglist:** Join the mailinglist by sending an email to [bottlepy@googlegroups.com](mailto:bottlepy@googlegroups.com) or using the web front-end at [google groups](#). You don't need a Google account to use the mailing-list, but then I have to approve your subscription manually to protect the list from spam. Please be patient in that case.
- **Twitter:** Follow us on '[Twitter<twitter.com/bottlepy>](#)' or search for the `#bottlepy` tag.
- **IRC:** Join `#bottlepy` on [irc.freenode.net](#).

### 3.2.2 Get the Sources

The bottle development repository and the issue tracker are both hosted at [github](#). If you plan to contribute, it is a good idea to create an account there and fork the main repository. This way your changes and ideas are visible to other developers and can be discussed openly. Even without an account, you can clone the repository using `git` (recommended) or just download the latest development version as a source archive.

- **git:** `git clone git://github.com/defnull/bottle.git`
- **git/http:** `git clone http://github.com/defnull/bottle.git`
- **svn:** `svn checkout http://svn.github.com/defnull/bottle.git` (not recommended)
- **Download:** Development branch as [tar archive](#) or [zip file](#).

### 3.2.3 Branches and their Intention

The bottle project follows a development and release workflow similar to the one described by [Vincent Driessen](#) in his [blog](#) (very interesting read) with some exceptions:

1. There is no `develop` branch. The `master` branch is used for development and integration.
2. Release-branches don't die. They stay alive as long as the release is still supported and host the most recent bug-fixed revision of that release.

- Hotfix-branches don't have to start with `hotfix-` but should end in a release number. Example: "some\_bug-0.8"

For a quick overview I'll describe the different branch-types and their intentions here:

**master** This is the integration, testing and development branch. All changes that are planned to be part of the next release are merged and tested here.

**release-x.y** As soon as the master branch is (almost) ready for a new release, it is branched into a new release branch. This "release candidate" is feature-frozen but may receive bug-fixes and last-minute changes until it is considered production ready and officially released. From that point on it is called a "support branch" and still receives bug-fixes, but only important ones. The revision number is increased on each push to these branches, so you can keep up with important changes.

**bugfix\_name-x.y** These branches are only temporary and used to develop and share non-trivial bug-fixes for existing releases. They are merged into the corresponding release branch and delete soon after that.

**Feature branches** All other branches are feature branches. These are based on the master branch and only live as long as they are still active and not merged back into `master`.

### What does this mean for a developer?

If you want to add a feature, create a new branch from `master`. If you want to fix a bug, branch `release-x.y` for each affected release. Please use a separate branch for each feature or bug to make integration as easy as possible. That's all. There are git workflow examples at the bottom of this page.

Oh, and never ever change the release number. I'll do that on integration. You never know in which order I am going to pull pending requests anyway :)

### What does this mean for a maintainer ?

Watch the tags (and the mailing list) for bug-fixes and new releases. If you want to fetch a specific release from the git repository, trust the tags, not the branches. A branch may contain changes that are not released yet, but a tag marks the exact commit which changed the version number.

### Patch Submission

The best way to get your changes integrated into the main development branch is to fork the main repository at github, create a new feature-branch, apply your changes and send a pull-request. Further down this page is a small collection of git workflow examples that may guide you. Submitting git-compatible patches to the mailing list is fine too. In any case, please follow some basic rules:

- **Documentation:** Tell us what your patch does. Comment your code. If you introduced a new feature, add to the documentation so others can learn about it.
- **Test:** Write tests to prove that your code works as expected and does not break anything. If you fixed a bug, write at least one test-case that triggers the bug. Make sure that all tests pass before you submit a patch.
- **One patch at a time:** Only fix one bug or add one feature at a time. Design your patches so that they can be applied as a whole. Keep your patches clean, small and focused.
- **Sync with upstream:** If the `upstream/master` branch changed while you were working on your patch, rebase or pull to make sure that your patch still applies without conflicts.

### 3.2.4 Releases and Updates

Bottle is released at irregular intervals and distributed through [PyPi](#). Release candidates and bugfix-revisions of outdated releases are only available from the git repository mentioned above. Some Linux distributions may offer packages for outdated releases, though.

The Bottle version number splits into three parts (**major.minor.revision**). These are *not* used to promote new features but to indicate important bug-fixes and/or API changes. Critical bugs are fixed in at least the two latest minor releases and announced in all available channels (mailinglist, twitter, github). Non-critical bugs or features are not guaranteed to be backported. This may change in the future, through.

**Major Release** The major release number is increased on important milestones or updates that completely break backward compatibility. You probably have to work over your entire application to use a new release. These releases are very rare, through.

**Minor Release** The minor release number is increased on updates that change the API or behaviour in some way. You might get some depreciation warnings any may have to tweak some configuration settings to restore the old behaviour, but in most cases these changes are designed to be backward compatible for at least one minor release. You should update to stay up do date, but don't have to. An exception is 0.8, which *will* break backward compatibility hard. (This is why 0.7 was skipped). Sorry for that.

**Revision** The revision number is increased on bug-fixes and other patches that do not change the API or behaviour. You can safely update without editing your application code. In fact, you really should as soon as possible, because important security fixes are released this way.

**Pre-Release Versions** Release candidates are marked by an `rc` in their revision number. These are API stable most of the time and open for testing, but not officially released yet. You should not use these for production.

### 3.2.5 GIT Workflow Examples

The following examples assume that you have an (free) account at [github](#). This is not mandatory, but makes things a lot easier.

First of all you have to create a fork (a personal clone) of the official repository. To do this, you simply click the “fork” button on the [bottle project page](#). When the fork is done, you will be presented with a short introduction to your new repository.

The fork you just created is hosted at github and read-able by everyone, but write-able only by you. Now you need to clone the fork locally to actually make changes to it. Make sure you use the private (read-write) URL and *not* the public (read-only) one:

```
git clone git@github.com:your_github_account/bottle.git
```

Once the clone is complete your repository will have a remote named “origin” that points to your fork on github. Don't let the name confuse you, this does not point to the original bottle repository, but to your own fork. To keep track of the official repository, add another remote named “upstream”:

```
cd bottle
git remote add upstream git://github.com/defnull/bottle.git
git fetch upstream
```

Note that “upstream” is a public clone URL, which is read-only. You cannot push changes directly to it. Instead, we will pull from your public repository. This is described later.

### Submit a Feature

New features are developed in separate feature-branches to make integration easy. Because they are going to be integrated into the `master` branch, they must be based on `upstream/master`. To create a new feature-branch, type the following:

```
git checkout -b cool_feature upstream/master
```

Now implement your feature, write tests, update the documentation, make sure that all tests pass and commit your changes:

```
git commit -a -m "Cool Feature"
```

If the `upstream/master` branch changed in the meantime, there may be conflicts with your changes. To solve these, ‘rebase’ your feature-branch onto the top of the updated `upstream/master` branch:

```
git fetch upstream
git rebase upstream
```

This is equivalent to undoing all your changes, updating your branch to the latest version and reapplying all your patches again. If you released your branch already (see next step), this is not an option because it rewrites your history. You can do a normal pull instead. Resolve any conflicts, run the tests again and commit.

Now you are almost ready to send a pull request. But first you need to make your feature-branch public by pushing it to your github fork:

```
git push origin cool_feature
```

After you’ve pushed your commit(s) you need to inform us about the new feature. One way is to send a pull-request using github. Another way would be to start a thread in the mailing-list, which is recommended. It allows other developers to see and discuss your patches and you get some feedback for free :)

If we accept your patch, we will integrate it into the official development branch and make it part of the next release.

### Fix a Bug

The workflow for bug-fixes is very similar to the one for features, but there are some differences:

1. Branch off of the affected release branches instead of just the development branch.
2. Write at least one test-case that triggers the bug.
3. Do this for each affected branch including `upstream/master` if it is affected. `git cherry-pick` may help you reducing repetitive work.
4. Name your branch after the release it is based on to avoid confusion. Examples: `my_bugfix-x.y` or `my_bugfix-dev`.

## 3.3 Plugin Development Guide

This guide explains the plugin API and how to write custom plugins. I suggest reading *Plugins* first if you have not done that already. You might also want to have a look at the [List of available Plugins](#) for some practical examples.

---

**Note:** This is a draft. If you see any errors or find that a specific part is not explained clear enough, please tell the [mailing-list](#) or file a [bug report](#).

---



### 3.3.1 How Plugins Work: The Basics

The plugin API builds on the concept of [decorators](#). To put it briefly, a plugin is a decorator that is applied to every single route of an application. Any decorator that can be applied to a route callback doubles as a plugin.

Of course, this is a simplification. Plugins can do a lot more than just decorating route callbacks, but it is a good starting point. Lets have a look at some code:

```
from bottle import response, install
import time

def stopwatch(callback):
    def wrapper(*args, **kwargs):
        start = time.time()
        rv = callback(*args, **kwargs)
        end = time.time()
        response.headers['X-Exec-Time'] = str(end - start)
        return rv
    return wrapper

bottle.install(stopwatch)
```

This plugin measures the execution time for each request and adds an appropriate `X-Exec-Time` header to the response. As you can see, the plugin returns a wrapper and the wrapper calls the original callback recursively. This is how decorators usually work.

The last line tells Bottle to install the plugin to the default application. This causes the plugin to be automatically applied to all routes of that application. In other words, `stopwatch()` is called once for each route callback and the return value is used as a replacement for the original callback.

Plugins are applied on demand, that is, as soon as a route is requested for the first time. For this to work properly in multi-threaded environments, the plugin needs to be thread-safe. This is not a problem most of the time, but keep it in mind.

Once all plugins are applied to a route, the prepared callback is cached and subsequent requests are handled by the cached version directly. This means that a plugin is usually applied only once to a specific route. That cache, however, is cleared every time the list of installed plugins changes. Your plugin should be able to decorate the same route more than once.

The decorator API is quite limited, though. You don't know anything about the route being decorated or the associated application object and have no way to efficiently store data that is shared among all routes. But fear not! Plugins are not limited to just decorator functions. Bottle accepts anything as a plugin as long as it is callable or implements an extended API. This API is described below and gives you a lot of control over the whole process.

### 3.3.2 Plugin API

*Plugin* is not a real class (you cannot import it from *bottle*) but an interface that plugins are expected to implement. Bottle accepts any object of any type as a plugin, as long as it conforms to the following API.

**class Plugin** (*object*)

Plugins must be callable or implement `apply()`. If `apply()` is defined, it is always preferred over calling the plugin directly. All other methods and attributes are optional.

**name**

Both `Bottle.uninstall()` and the `skip` parameter of `Bottle.route()` accept a name string to refer to a plugin or plugin type. This works only for plugins that have a name attribute.

**setup** (*app*)

Called as soon as the plugin is installed to an application (see `Bottle.install()`). The only parameter

is the associated application object. This method is *not* called on plugins that are applied directly to routes via the `Bottle.route()` decorator.

**\_\_call\_\_**(*callback*)

As long as `apply()` is not defined, the plugin itself is used as a decorator and applied directly to each route callback. The only parameter is the callback to decorate. Whatever is returned by this method replaces the original callback. If there is no need to wrap or replace a given callback, just return the unmodified callback parameter.

**apply**(*callback, context*)

If defined, this method is used instead of `__call__()` to decorate route callbacks. The additional context parameter is a dictionary that contains any keyword arguments passed to the `Bottle.route()` decorator, as well as some additional meta-information about the route being decorated. See [Route Context](#) for details.

**close**()

Called immediately before the plugin is uninstalled or the application is closed (see `Bottle.uninstall()` or `Bottle.close()`). This method is *not* called on plugins that are applied directly to routes via the `route()` decorator.

### 3.3.3 Route Context

The route context dictionary stores meta-information about a specific route. It is passed to `Plugin.apply()` along with the route callback and contains the following values:

Key	Description
rule	The rule string (e.g. <code>/wiki/:page</code> ) as it is passed to the router.
method	An uppercase HTTP method string (e.g. <code>GET</code> ).
call-back	The original callback with no plugins or wrappers applied. Useful for introspection.
name	The name of the route (if specified) or <code>None</code> .
ap- ply	A list of route-specific plugins (see <code>Bottle.route()</code> ).
skip	A list of plugins to not apply to this route (see <code>Bottle.route()</code> ).
app	The associated application object.
con- fig	Additional keyword arguments passed to the <code>Bottle.route()</code> decorator are stored in this dictionary. Used for route-specific plugin configuration and meta-data.
id	An internal handle used by bottle to identify a route.

The `Bottle.route()` decorator accepts multiple rules and methods in a single call, but the context dictionary refers to a specific pair only. `Plugin.apply()` is called once for each combination of `rule` and `method`, even if they all map to the same route callback.

Keep in mind that the `config` dictionary is shared between all plugins. It is always a good idea to add a unique prefix or, if your plugin needs a lot of configuration, store it in a separate dictionary within the `config` dictionary. This helps to avoid naming collisions between plugins.

### Manipulating the Context Dictionary

While the *route context dictionary* is mutable, changes may have unpredictable effects on other plugins. It is most likely a bad idea to monkey-patch a broken configuration instead of providing a helpful error message and let the user fix it properly.

In some rare cases, however, it might be justifiable to break this rule. After you made your changes to the context dictionary, raise `RouteReset` as an exception. This removes the current route from the callback cache and causes

all plugins to be re-applied. The router is not updated, however. Changes to *rule* or *method* values have no effect on the router, but only on plugins. This may change in the future, though.

### 3.3.4 Plugin Example: SQLitePlugin

This plugin provides an sqlite3 database connection handle as an additional keyword argument to wrapped callbacks, but only if the callback expects it. If not, the route is ignored and no overhead is added. The wrapper does not affect the return value, but handles plugin-related exceptions properly. *Plugin.setup()* is used to inspect the application and search for conflicting plugins.

```
import sqlite3
import inspect

class SQLitePlugin(object):
    ''' This plugin passes an sqlite3 database handle to route callbacks
    that accept a `db` keyword argument. If a callback does not expect
    such a parameter, no connection is made. You can override the database
    settings on a per-route basis. '''

    name = 'sqlite'

    def __init__(self, dbfile=':memory:', autocommit=True, dictrows=True,
                 keyword='db'):
        self.dbfile = dbfile
        self.autocommit = autocommit
        self.dictrows = dictrows
        self.keyword = keyword

    def setup(self, app):
        ''' Make sure that other installed plugins don't affect the same
        keyword argument. '''
        for other in app.plugins:
            if not isinstance(other, SQLitePlugin): continue
            if other.keyword == self.keyword:
                raise PluginError("Found another sqlite plugin with "\
                                 "conflicting settings (non-unique keyword).")

    def apply(self, callback, context):
        # Override global configuration with route-specific values.
        conf = context['config'].get('sqlite') or {}
        dbfile = conf.get('dbfile', self.dbfile)
        autocommit = conf.get('autocommit', self.autocommit)
        dictrows = conf.get('dictrows', self.dictrows)
        keyword = conf.get('keyword', self.keyword)

        # Test if the original callback accepts a 'db' keyword.
        # Ignore it if it does not need a database handle.
        args = inspect.getargspec(context['callback'])[0]
        if keyword not in args:
            return callback

    def wrapper(*args, **kwargs):
        # Connect to the database
        db = sqlite3.connect(dbfile)
        # This enables column access by name: row['column_name']
        if dictrows: db.row_factory = sqlite3.Row
        # Add the connection handle as a keyword argument.
```

```
kwargs[keyword] = db

    try:
        rv = callback(*args, **kwargs)
        if autocommit: db.commit()
    except sqlite3.IntegrityError, e:
        db.rollback()
        raise HTTPError(500, "Database Error", e)
    finally:
        db.close()
    return rv

# Replace the route callback with the wrapped one.
return wrapper
```

This plugin is actually useful and very similar to the version bundled with Bottle. Not bad for less than 60 lines of code, don't you think? Here is a usage example:

```
sqlite = SQLitePlugin(dbfile='/tmp/test.db')
bottle.install(sqlite)

@route('/show/:page')
def show(page, db):
    row = db.execute('SELECT * from pages where name=?', page).fetchone()
    if row:
        return template('showpage', page=row)
    return HTTPError(404, "Page not found")

@route('/static/:fname#.*#')
def static(fname):
    return static_file(fname, root='/some/path')

@route('/admin/set/:db#[a-zA-Z]+#', skip=[sqlite])
def change_dbfile(db):
    sqlite.dbfile = '/tmp/%s.db' % db
    return "Switched DB to %s.db" % db
```

The first route needs a database connection and tells the plugin to create a handle by requesting a `db` keyword argument. The second route does not need a database and is therefore ignored by the plugin. The third route does expect a `db` keyword argument, but explicitly skips the `sqlite` plugin. This way the argument is not overruled by the plugin and still contains the value of the same-named url argument.

## 3.4 List of available Plugins

The plugin API is extremely flexible and most plugins are designed to be portable and re-usable across applications. Below is an incomplete list of available plugins, all tested with the latest version of bottle and ready to use.

Have a look at [Plugins](#) for general questions about plugins (installation, usage). If you plan to develop a new plugin, the [Plugin Development Guide](#) may help you.

- **Bottle-SQLite**: Adds support for *SQLite* databases.
- **Bottle-Werkzeug**: Integrates the *werkzeug* library (alternative request and response objects, advanced debugging middleware and more).
- **profile**: This plugin collects profiling data and displays it in the browser.

### 3.4.1 Bottle-SQLite

SQLite is a self-contained SQL database engine that runs locally and does not require any additional server software or setup. The `sqlite3` module is part of the Python standard library and already installed on most systems. It is very useful for prototyping database-driven applications that are later ported to larger databases such as PostgreSQL or MySQL.

This plugin simplifies the use of sqlite databases in your Bottle applications. Once installed, all you have to do is to add a `db` keyword argument (configurable) to route callbacks that need a database connection.

#### Installation

Install with one of the following commands:

```
$ pip install bottle-sqlite
$ easy_install bottle-sqlite
```

or download the latest version from github:

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/sqlite
$ python setup.py install
```

#### Usage

Once installed to an application, the plugin passes an open `sqlite3.Connection` instance to all routes that require a `db` keyword argument:

```
import bottle

app = bottle.Bottle()
plugin = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
app.install(plugin)

@app.route('/show/:item')
def show(item, db):
    row = db.execute('SELECT * from items where name=?', item).fetchone()
    if row:
        return template('showitem', page=row)
    return HTTPError(404, "Page not found")
```

Routes that do not expect a `db` keyword argument are not affected.

The connection handle is configured so that `sqlite3.Row` objects can be accessed both by index (like tuples) and case-insensitively by name. At the end of the request cycle, outstanding transactions are committed and the connection is closed automatically. If an error occurs, any changes to the database since the last commit are rolled back to keep the database in a consistent state.

#### Configuration

The following configuration options exist for the plugin class:

- **dbfile**: Database filename (default: in-memory database).
- **keyword**: The keyword argument name that triggers the plugin (default: `'db'`).
- **autocommit**: Whether or not to commit outstanding transactions at the end of the request cycle (default: `True`).

- **dictrows**: Whether or not to support dict-like access to row objects (default: True).

You can override each of these values on a per-route basis:

```
@app.route('/cache/:item', sqlite={'dbfile': ':memory:'})
def cache(item, db):
    ...
```

or install two plugins with different keyword settings to the same application:

```
app = bottle.Bottle()
test_db = bottle.ext.sqlite.Plugin(dbfile='/tmp/test.db')
cache_db = bottle.ext.sqlite.Plugin(dbfile=':memory:', keyword='cache')
app.install(test_db)
app.install(cache_db)

@app.route('/show/:item')
def show(item, db):
    ...

@app.route('/cache/:item')
def cache(item, cache):
    ...
```

### 3.4.2 Bottle-Werkzeug

**Werkzeug** is a powerfull WSGI utility library for Python. It includes an interactive debugger and feature-packed request and response objects.

This plugin integrates `werkzeug.wrappers.Request` and `werkzeug.wrappers.Response` as an alternative to the built-in implementations, adds support for `werkzeug.exceptions` and replaces the default error page with an interactive debugger.

#### Installation

Install with one of the following commands:

```
$ pip install bottle-werkzeug
$ easy_install bottle-werkzeug
```

or download the latest version from github:

```
$ git clone git://github.com/defnull/bottle.git
$ cd bottle/plugins/werkzeug
$ python setup.py install
```

#### Usage

Once installed to an application, this plugin adds support for `werkzeug.wrappers.Response`, all kinds of `werkzeug.exceptions` and provides a thread-local instance of `werkzeug.wrappers.Request` that is updated with each request. The plugin instance itself doubles as a `werkzeug` module object, so you don't have to import `werkzeug` in your application. Here is an example:

```
import bottle

app = bottle.Bottle()
```

```
werkzeug = bottle.ext.werkzeug.Plugin()
app.install(werkzeug)

req = werkzeug.request # For the lazy.

@app.route('/hello/:name')
def say_hello(name):
    greet = {'en': 'Hello', 'de': 'Hallo', 'fr': 'Bonjour'}
    language = req.accept_languages.best_match(greet.keys())
    if language:
        return werkzeug.Response('%s %s!' % (greet[language], name))
    else:
        raise werkzeug.exceptions.NotAcceptable()
```

## Using the Debugger

This plugin replaces the default error page with an advanced debugger. If you have the *evalex* feature enabled, you will get an interactive console that allows you to inspect the error context in the browser. Please read Debugging Applications with werkzeug before you enable this feature.

## Configuration

The following configuration options exist for the plugin class:

- **evalex**: Enable the exception evaluation feature (interactive debugging). This requires a non-forking server and is a security risk. Please read Debugging Applications with werkzeug. (default: False)
- **request\_class**: Defaults to `werkzeug.wrappers.Request`
- **debugger\_class**: Defaults to a subclass of `werkzeug.debug.DebuggedApplication` which obeys the `bottle.DEBUG` setting.





---

### License

---

Code and documentation are available according to the MIT License:

```
Copyright (c) 2011, Marcel Hellkamp.
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy  
of this software and associated documentation files (the "Software"), to deal  
in the Software without restriction, including without limitation the rights  
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell  
copies of the Software, and to permit persons to whom the Software is  
furnished to do so, subject to the following conditions:
```

```
The above copyright notice and this permission notice shall be included in  
all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR  
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,  
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE  
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER  
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,  
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN  
THE SOFTWARE.
```

The Bottle logo however is *NOT* covered by that license. It is allowed to use the logo as a link to the bottle homepage or in direct context with the unmodified library. In all other cases please ask first.



**b**

bottle, 3



## Symbols

`__call__()` (Plugin method), 62  
`__init__()` (BaseTemplate method), 30

## A

`app()` (in module bottle), 24  
`apply()` (Plugin method), 62  
AppStack (class in bottle), 24, 26  
`auth` (Request attribute), 29

## B

BaseTemplate (class in bottle), 30  
`bind()` (Request method), 28  
`bind()` (Response method), 29  
Bottle (class in bottle), 26  
bottle (module), 3, 23, 31, 51, 54, 60, 64  
BottleException, 26

## C

callback, 19  
catchall (Bottle attribute), 26  
cgikeys (WSGIHeaderDict attribute), 26  
charset (Response attribute), 29  
`close()` (Bottle method), 27  
`close()` (Plugin method), 62  
content\_length (Request attribute), 28  
content\_type (Response attribute), 30  
cookie\_decode() (in module bottle), 25  
cookie\_encode() (in module bottle), 25  
cookie\_is\_encoded() (in module bottle), 25  
COOKIES (Request attribute), 29  
COOKIES (Response attribute), 29  
`copy()` (Request method), 28  
`copy()` (Response method), 29

## D

`debug()` (in module bottle), 24  
decorator, 20  
`default_app()` (in module bottle), 24  
`delete()` (Bottle method), 27

`delete()` (in module bottle), 25  
`delete_cookie()` (Response method), 30

## E

environ, 20  
`error()` (Bottle method), 27  
`error()` (in module bottle), 25

## F

files (Request attribute), 29  
forms (Request attribute), 29  
fullpath (Request attribute), 28

## G

GET (Request attribute), 29  
`get()` (Bottle method), 27  
`get()` (in module bottle), 25  
`get_content_type()` (Response method), 30  
`get_cookie()` (Request method), 29  
`get_url()` (Bottle method), 27  
`global_config()` (bottle.BaseTemplate class method), 30

## H

`handle()` (Bottle method), 28  
handler function, 20  
HeaderDict (class in bottle), 25  
headerlist (Response attribute), 29  
headers (Request attribute), 28  
`hook()` (Bottle method), 28  
HTTP\_CODES (in module bottle), 24  
HTTPError, 26  
HTTPResponse, 26

## I

`install()` (Bottle method), 26  
`is_ajax` (Request attribute), 29

## L

`load_app()` (in module bottle), 24

## M

`match()` (Bottle method), [27](#)  
`mount()` (Bottle method), [26](#)  
`MultiDict` (class in bottle), [25](#)

## N

`name` (Plugin attribute), [61](#)

## P

`params` (Request attribute), [29](#)  
`parse_auth()` (in module bottle), [25](#)  
`parse_date()` (in module bottle), [25](#)  
`path_shift()` (in module bottle), [25](#)  
`path_shift()` (Request method), [28](#)  
`Plugin` (class in bottle), [61](#)  
`pop()` (AppStack method), [24](#)  
`POST` (Request attribute), [29](#)  
`post()` (Bottle method), [27](#)  
`post()` (in module bottle), [25](#)  
`prepare()` (BaseTemplate method), [30](#)  
`push()` (AppStack method), [24](#), [26](#)  
`put()` (Bottle method), [27](#)  
`put()` (in module bottle), [25](#)

## Q

`query_string` (Request attribute), [28](#)

## R

`raw()` (WSGIHeaderDict method), [26](#)  
`render()` (BaseTemplate method), [31](#)  
`render()` (SimpleTemplate method), [23](#)  
`Request` (class in bottle), [28](#)  
`request` (in module bottle), [24](#)  
`reset()` (Bottle method), [27](#)  
`Response` (class in bottle), [29](#)  
`response` (in module bottle), [24](#)  
`route()` (Bottle method), [27](#)  
`route()` (in module bottle), [25](#)  
`RouteReset`, [26](#)  
`run()` (in module bottle), [24](#)

## S

`search()` (bottle.BaseTemplate class method), [30](#)  
`set_cookie()` (Response method), [29](#)  
`setup()` (Plugin method), [61](#)  
`SimpleTemplate` (class in bottle), [23](#)  
`source directory`, [20](#)  
`split_comment()` (bottle.SimpleTemplate class method),  
[23](#)

## T

`template()` (in module bottle), [31](#)

## U

`uninstall()` (Bottle method), [27](#)  
`url` (Request attribute), [28](#)  
`urlparts` (Request attribute), [28](#)

## V

`view()` (in module bottle), [31](#)

## W

`wsgi()` (Bottle method), [28](#)  
`wsgiheader()` (Response method), [29](#)  
`WSGIHeaderDict` (class in bottle), [26](#)

## Y

`yieldroutes()` (in module bottle), [25](#)