Bottle Documentation

Release 0.9.dev

Marcel Hellkamp

October 10, 2010

CONTENTS

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, 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 [1] dependencies other than the Python standard library. Bottle runs with **Python 2.5+ and 3.x** (using 2to3)

[1] Usage of the template or server adapter classes of course requires the corresponding template or server modules.

CHAPTER

ONE

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 Tutorial

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.

Note: This is a copy&paste from the old docs and a work in progress. Handle with care :)

A quick overview:

- *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 write HTML within your python code, do you? Templates separate code from presentation.
- *Development*: These tools and features will help you during development.
- *Deployment*: Get it up and running.

1.1.1 Getting started

Bottle has no dependencies, so all you need is Python (2.5 up to 3.x should work fine) and the *bottle module* file. Lets 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)
```

Whats happening here?

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

This is it. Run this script, visit http://localhost:8080/hello and you will see "Hello World!" in your browser. Of cause 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

Several functions and decorators such as `route()` or `run()` rely on a global application object to store routes, callbacks and configuration. This makes writing a small application easy, but can lead to problems in more complex scenarios. If you prefer a more explicit way to define your application and don't mind the extra typing, you can create your own concealed application object and use that instead of the global one:

```
from bottle import Bottle, run

myapp = Bottle()

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

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

This tutorial uses the global-application syntax for the sake of simplicity. Just keep in mind that you have a choice. The object-oriented approach is further described in the *tutorial-appobject* section.

1.1.2 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" />
              </from>'''

#@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 error pages using the `error()` decorator. It works similar to the `route()` decorator but expects an HTTP status code instead of a route:

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

The `error` parameter passed to the error handler is an instance of `HTTPError`.

1.1.3 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 *tutorial-errorhandling* 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

TODO

Secure Cookies

TODO

1.1.4 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 do 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! Your 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
```

1.1.5 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.

1.1.6 Development

Bottle has two features that may be helpful during development.

Debug Mode

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

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

This does the following:

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

Auto Reloading

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

```
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 respawned by the main process. Changes in template files will not trigger a reload. Please use debug mode to deactivate template caching.

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

1.1.7 Deployment

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

There are three ways to eliminate this bottleneck:

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

Multi-Threaded Server

The easiest way to increase performance is to install a multi-threaded and WSGI-capable HTTP server like Paste, flup, cherrypy or fapws3 and use the corresponding bottle server-adapter.

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

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

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

Multiple Server Processes

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

Instead of a single Bottle application server, you start one 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.

But there are a few drawbacks:

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

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

I'll add examples for lighttpd and Apache web servers soon.

Using WSGI and Middleware

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

```
from bottle import default_app, run
app = default_app()
newapp = YourMiddleware(app)
run(app=newapp)
```

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

```
from bottle import Bottle, run
mybottle = Bottle()
@mybottle.route('/')
def index():
  return 'default_app'
run(app=mybottle)
```

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
# ... add or import your bottle app code 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

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

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

Good old CGI

CGI is slow as hell, but it works:

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

1.1.8 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.

secure cookie Bottle creates signed cookies with objects that can be pickled. A secure cookie will be created automatically when a type that is not a string is used as the value in `request.set_cookie()` and bottle's config includes a *securecookie.key* entry with a salt.

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 foother_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*)

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 Frequently Asked Questions

1.3.1 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.

1.3.2 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.

CHAPTER

TWO

API DOCUMENTATION

Looking for a specific function, class or method? These chapters cover all the interfaces provided by the Framework and explain how to use them.

2.1 API Reference

Platforms: Unix, Windows This is an API reference, NOT documentation. If you are new to bottle, have a look at the *Tutorial*.

2.1.1 Module Contents

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

`app`()
Return the current *default application* (see `Bottle`). Actually, this is a callable instance of `AppStack` and implements a stack-like API.

`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*, ***kargs*)
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)
- **host** – 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
Whenever a page is requested, the `Bottle` WSGI handler stores metadata about the current request into this instance of `Request`. It is thread-safe and can be accessed from within handler functions.

response
The `Bottle` WSGI handler uses metadata assigned to this instance of `Response` to generate the WSGI response.

HTTP_CODES
dict() -> new empty dictionary. dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs.

dict(seq) -> new dictionary initialized as if via: d = {} for k, v in seq:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

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'*, *name=None*)
Decorator to bind a function to a path. This equals `Bottle.route()` using the current default application.

get (...)
post (...)
put (...)
delete (...)
These are equal to `route()` with the *method* parameter set to the corresponding verb.

error (...)
Calls `Bottle.error()` using the default application.

url (...)
Calls `Bottle.url()` using the default application.

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 by default.

class **WSGIHeaderDict**(*environ*)
This dict-like class takes a WSGI environ dict and provides convenient access to HTTP_* fields. Keys and values are stored as native strings (bytes/unicode) based on the python version used (2/3) and keys are case-insensitive. If the WSGI environment contains non-native strings, these are de- or encoded using 'utf8' (default) or 'latin1' (fallback) charset. To get the original value, use the .raw(key) method.

This is not a MultiDict because incoming headers are unique. The API will remain stable even on WSGI spec changes, if possible.

raw(*key*, *default=None*)
Return the raw WSGI header value for that key.

class **AppStack**
A stack implementation.

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

2.1.2 The `Bottle` Class

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

`add_filter` (*ftype*, *func*)
Register a new output filter. Whenever bottle hits a handler output matching *ftype*, *func* is applied to it.

`add_hook` (*name*, *func*)
Add a callback from a hook.

`delete` (*path=None*, *method='DELETE'*, ***kargs*)
Decorator: Bind a function to a DELETE request path. See :meth:'route' for details.

`error` (*code=500*)
Decorator: Register an output handler for a HTTP error code

`get` (*path=None*, *method='GET'*, ***kargs*)
Decorator: Bind a function to a GET request path. See :meth:'route' for details.

`get_url` (*routename*, ***kargs*)
Return a string that matches a named route

`handle` (*url*, *method*)
Execute the handler bound to the specified url and method and return its output. If catchall is true, exceptions are catched and returned as HTTPError(500) objects.

`hook` (*name*)
Return a decorator that adds a callback to the specified hook.

`match_url` (*path*, *method='GET'*)
Find a callback bound to a path and a specific HTTP method. Return (callback, param) tuple or raise HTTPError. method: HEAD falls back to GET. All methods fall back to ANY.

`mount` (*app*, *script_path*)
Mount a Bottle application to a specific URL prefix

`post` (*path=None*, *method='POST'*, ***kargs*)
Decorator: Bind a function to a POST request path. See :meth:'route' for details.

`put` (*path=None*, *method='PUT'*, ***kargs*)
Decorator: Bind a function to a PUT request path. See :meth:'route' for details.

`remove_hook` (*name*, *func*)
Remove a callback from a hook.

`route` (*path=None*, *method='GET'*, *no_hooks=False*, *decorate=None*, *template=None*, *template_opts={}*, *callback=None*, ***kargs*)
Decorator: Bind a callback function to a request path.

Parameters

- **path** – The request path or a list of paths to listen to. See `Router` for syntax details. If no path is specified, it is automatically generated from the callback signature. See `yieldroutes()` for details.
- **method** – The HTTP method (POST, GET, ...) or a list of methods to listen to. (default: GET)

- **decorate** – A decorator or a list of decorators. These are applied to the callback in reverse order.
- **no_hooks** – If true, application hooks are not triggered by this route. (default: False)
- **template** – The template to use for this callback. (default: no template)
- **template_opts** – A dict with additional template parameters.
- **static** – If true, all paths are static even if they contain dynamic syntax tokens. (default: False)
- **name** – The name for this route. (default: None)
- **callback** – If set, the route decorator is directly applied to the callback and the callback is returned instead. This equals `Bottle.route(...)(callback)`.

`wsgi`(*environ*, *start_response*)
The bottle WSGI-interface.

2.1.3 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.

`COOKIES`
Cookies parsed into a dictionary. Secure cookies are NOT decoded automatically. See `get_cookie()` for details.

`GET`
The QUERY_STRING parsed into an instance of `MultiDict`.

If you expect more than one value for a key, use `.getall(key)` on this dictionary to get a list of all values. Otherwise, only the first value is returned.

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

If you expect more than one value for a key, use `.getall(key)` on this dictionary to get a list of all values. Otherwise, only the first value is returned.

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

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

`bind`(*environ*)
Bind a new WSGI environment.

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

body
The HTTP request body as a seekable file-like object.

This property returns a copy of the *wsgi.input* stream and should be used instead of *environ['wsgi.input']*.

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

copy()
Returns a copy of self

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`.

If you expect more than one value for a key, use `.getall(key)` on this dictionary to get a list of all values. Otherwise, only the first value is returned.

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 bidy. The values are native strings.

If you expect more than one value for a key, use `.getall(key)` on this dictionary to get a list of all values. Otherwise, only the first value is returned.

fullpath
Request path including SCRIPT_NAME (if present).

get_cookie(*key*, *secret=None*)
Return the content of a cookie. To read a *Secure Cookies*, use the same *secret* as used to create the cookie (see `Response.set_cookie()`). If anything goes wrong, None is returned.

headers
A dict-like object filled with request headers.

This dictionary uses case-insensitive keys and native strings as keys and values. See `WSGIHeaderDict` for details.

is_ajax
True if the request was generated using XMLHttpRequest

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

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)

query_string
The part of the URL following the '?'.

url
Full URL as requested by the client (computed).

This value is constructed out of different environment variables and includes scheme, host, port, scriptname, path and query string.

class **`Response`**
Represents a single HTTP response using thread-local attributes.

`COOKIES`
A dict-like SimpleCookie instance. Use `set_cookie()` instead.

`bind``()`
Resets the Response object to its factory defaults.

`charset`
Return the charset specified in the content-type header.

This defaults to *UTF-8*.

`content_type`
Current 'Content-Type' header.

`copy``()`
Returns a copy of self.

`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.

`headerlist`
Returns a wsgi conform list of header/value pairs.

`set_cookie``(`*key*, *value*, *secret=None*, ***kargs*`)`
Add a cookie. If the *secret* parameter is set, this creates a *Secure Cookie* (described below).

Parameters

- **key** – the name of the cookie.
- **value** – the value of the cookie.
- **secret** – required for secure cookies. (default: None)
- **max_age** – maximum age in seconds. (default: None)
- **expires** – a datetime object or UNIX timestamp. (defaut: 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.

Secure 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: Secure 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.

`wsgiheader``()`
Returns a wsgi conform list of header/value pairs.

2.1.4 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 **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).

classmethod **search**(*name, lookup=[]*)
Search name in all directories specified in lookup. First without, then with common extensions. Return first hit.

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
`SimpleTemplate`	*SimpleTemplate Engine*	`view()`	`template()`
`MakoTemplate`	http://www.makotemplates.org	`mako_view()`	`mako_template()`
`CheetahTemplate`	http://www.cheetahtemplate.org/	`cheetah_view()`	`cheetah_template()`
`Jinja2Template`	http://jinja.pocoo.org/	`jinja2_view()`	`jinja2_template()`

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
```

CHAPTER

THREE

TUTORIALS AND RESOURCES

3.1 Tutorial: Todo-List Application

Note: This tutorial is a work in progess 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 databse, 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

3.1.1 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

3.1.2 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 NUL
con.execute("INSERT INTO todo (task,status) VALUES ('Read A-byte-of-python to get a good introductio
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 h
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).

3.1.3 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 quiet 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 quiet 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 ite
<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 `@valdiate` 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:

```python
@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 a 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.

3.1.4 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 servse 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=PasterServer)
```

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`

3.1.5 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 .

3.1.6 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 ite
<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>
```

3.2 Recipes

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

3.2.1 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)
```

3.2.2 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**.

3.2.3 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.

3.2.4 Ignore trailing slashes

For Bottle, `/example` and `/example/` are two different routes. 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:

```
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)
```

CHAPTER

FOUR

DEVELOPMENT AND CONTRIBUTION

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

4.1 Release Notes and Changelog

4.1.1 Release 0.9

This changes are not released yet and are only part of the development documentation.

New Features

- A new hook-API to inject code immediately before or after the execution of handler callbacks.
- The `Bottle.route()` decorator got a lot of new features. See API documentation for details.
- The `Request.headers` dict is now guaranteed to contain native strings but still allows access to the raw data provided by the WSGI environment (see `WSGIHeaderDict`).

API changes

- `Request.header` is now `Request.headers`

4.1.2 Bugfix Release 0.8.3

- Fixed "Reloading server dies on slow hardware." (Issue #90)

4.1.3 Bugfix Release 0.8.2

- Added backward compatibility wrappers and deprecation warnings to some of the API changes.
- Fixed "FileCheckerThread seems to fail on eggs" (Issue #87)
- Fixed "Bottle.get_url() does not return correct path when SCRIPT_NAME is set." (Issue #83)

4.1.4 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.

4.2 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!

4.2.1 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 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.

4.2.2 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.

4.2.3 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.
3. 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. Thats 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 applyed 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.

4.2.4 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.

4.2.5 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`.

4.3 Plugin Development

Bottles core features cover most of the common use-cases, but as a micro-framework it has its limits. This is where "Plugins" come into play. Plugins add specific functionality to the framework in a convenient way and are portable and re-usable across applications. Browse the list of `available plugins` and see if someone has solved your problem already. If not, then read ahead. This guide explains the use of middleware, decorators and the hook-api that makes writing plugins for bottle a snap.

4.3.1 Best Practice

These rules are recommendations only, but following them makes it a lot easier for others to use your plugin.

Initializing Plugins

Importing a plugin-module should not have any side-effects and particularly **never install the plugin automatically**. Instead, plugins should define a class or a function that handles initialization:

The class-constructor or init-function should accept an instance of `Bottle` as its first optional keyword argument and install the plugin to that application. If the *app*-argument is empty, the plugin should default to `default_app()`. All other arguments are specific to the plugin and optional.

For consistency, function-names should start with "*init_*" and class-names should end in "*Plugin*". It is ok to add an `init_*` alias for a class, but the class itself should conform to PEP8. Example:

```
import bottle

def init_myfeature(app=None):
    if not app:
        app = bottle.default_app()

    @app.hook('before_request')
    def before_hook():
        pass

class MyFeaturePlugin(object):
    def __init__(app=None):
        self.app = app or bottle.default_app()
        self.app.add_hook('before_request', self.before_hook)

    def before_hook():
        pass
```

Plugin Configuration

Plugins should use the `Bottle.config` dictionary and the `plugin.[name]` namespace for their configuration. This way it is possible to pre-configure plugins or change the configuration at runtime in a plugin-independent way. Example:

```
import bottle

class MyFeaturePlugin(object):
    def __init__(app=None):
        self.app = app or bottle.default_app()
        self.app.add_hook('before_request', self.before_hook)

    def before_hook():
        value = self.app.config.get('plugins.myfeature.key', 'default')
        ...
```

WSGI Middleware

WSGI middleware should not wrap the entire application object, but only the `Bottle.wsgi()` method. This way the app object stays intact and more than one middleware can be applied without conflicts.

4.3.2 Writing Plugins

In most cases, plug-ins are used to alter the the request/response circle in some way. They add, manipulate or remove information from the request and/or alter the data returned to the browser. Some plug-ins do not touch the request itself, but have other side effects such as opening and closing database connections or cleaning up temporary files. Apart from that, you can differentiate plug-ins by the point of contact with the application:

Middleware WSGI-middleware wraps an entire application. It is an application itself and calls the wrapped application internally. This way a middleware can alter both the incoming environ-dict and the response iterable before it is returned to the server. This is transparent to the wrapped application and does not require any special support or preparation. The downside of this approach is that the request and response objects are both not available and you have to deal with raw WSGI.

Decorators The decorator approach is best for wrapping a small number of routes while leaving all other callbacks untouched. If your application requires session support or database connections for only some of the routes, choose this approach. With a decorator you have full access to the request and response objects and the unfiltered return value of the wrapped callback.

Hooks With *hooks* you can register functions to be called at specific stages during the request circle. The most interesting hooks are *before_request* and *after_request*. Both affect all routes in an application, have full control over the request and response objects and can manipulate the route-callback return value at will. This new API fills the gap between middleware and decorators and is described in detail further down this guide.

Which technique is best for your plugin depends on the level and scope of interaction you need with the framework and application. Combinations are possible, too. The following table sums it up:

Aspect	Middleware	Hooks	Decorators
Affects whole application	Yes	Yes	No
Access to Bottle features	No	Yes	Yes

Writing Middleware

WSGI middleware is not specific to Bottle and there are several detailed explanations and collections available. If you want to apply a WSGI middleware, wrap the `Bottle` application object and you're done:

```
app = bottle.app()            # Get the WSGI callable from bottle
app = MyMiddleware(app=app)  # Wrap it
bottle.run(app)               # Run it
```

This approach works fine, but is not very portable (see *Best Practice*). A more general approach is to define a function that takes care of the plugin initialization and keeps the original application object intact:

```
import bottle
def init_my_middleware(app=None, **config):
    # Default to the global application object
    if not app:
        app = bottle.app()
    # Do not wrap the entire application, but only the WSGI part
    app.wsgi = MyMiddleware(app=app.wsgi, config=config)
```

Now `app` is still an instance of `Bottle` and all methods remain accessible. Other plugins can wrap `app.wsgi` again without any conflicts.

Writing Decorators

Bottle uses decorators all over the place, so you should already now how to use them. Writing a decorator (or a decorator factory, see below) is not that hard, too. Basically a decorator is a function that takes a function object and returns either the same or a new function object. This way it is possible to *wrap* a function and alter its input and output whenever that function gets called. Decorators are an extremely flexible way to reduce repetitive work:

```
from bottle import route

def integer_id(func):
    ''' Make sure that the ``id`` keyword argument is an integer. '''
    def wrapper(*args, **kwargs):
        if 'id' in kwargs and not isinstance(kwargs['id'], int):
            kwargs['id'] = int(kwargs['id'])
        return func(*args, **kwargs)
    return wrapper
```

```
@route('/get/:id#[0-9]+#')
@integer_id
def get_object(id, ...):
    ...
```

Note: Decorators are applied in reverse order (the decorator closest to the 'def' statement is applied first). This is important if you want to apply more than one decorator.

Decorator factories: Configurable decorators

Let's go one step further: A *decorator factory* is a function that return a decorator. Because inner functions have access to the local variables of the outer function they were defined in, we can use this to configure the behavior of our decorator. Here is an example:

```
from bottle import request, response, abort

def auth_required(users, realm='Secure Area'):
    def decorator(func):
        def wrapper(*args, **kwargs):
            name, password = request.auth()
            if name not in users or users[name] != password:
                response.headers['WWW-Authenticate'] = 'Basic realm="%s"' % realm
                abort('401', 'Access Denied. You need to login first.')
            kwargs['user'] = name
            return func(*args, **kwargs)
        return wrapper
    return decorator

@route('/secure/area')
@auth_required(users={'Bob':'1234'})
def secure_area(user):
    print 'Hello %s' % user
```

Of cause it is a bad idea to store clear passwords in a dictionary. But besides that, this example is actually quite complete and usable as it is.

Using Hooks

New in version 0.9. As described above, hooks allow you to register functions to be called at specific stages during the request circle. There are currently only two hooks available:

before_request This hook is called immediately before each route callback.

after_request This hook is called immediately after each route callback.

You can use the `hook()` or `Bottle.hook()` decorator to register a function to a hook. This example shows how to open and close a database connection (SQLite 3) with each request:

```
import sqlite3
import bottle

def init_sqlite(app=None, dbfile=':memory:'):
    if not app:
        app = bottle.app()

    @app.hook('before_request')
    def before_request():
        bottle.local.db = sqlite3.connect(dbfile)

    @app.hook('after_request')
    def after_request():
        bottle.local.db.close()
```

The `local` object is used to store the database handle during the request. It is a thread-save object (just like `request` and `response` are) even if it looks like a global module variable. Here is an example for an application using this plugin:

```
from bottle import default_app, local, route, run
from plugins.sqlite import init_sqlite # Or whatever you named your plugin

@route('/wiki/:name')
@view('wiki_page')
def show_page(name):
    sql = 'select title, text rom wiki_pages where name = ?'
    cursor = local.db.execute(sql, name)
    entry = cursor.fetch()
    return dict(name=name, title=entry[0], text=entry[1])

init_sqlite(dbfile='wiki.db') # Install plugin to default app

if __name__ == '__main__':
    run() # Run default app
```

Plugin Classes

The problem with the last example is that you cannot access the plugin or the database object outside of a running server instance. Let's rewrite the plugin and use a class this time:

```
import sqlite3
import bottle

class SQlitePlugin(object):
    def __init__(self, app=None, dbfile=':memory:'):
        self.app = app or bottle.app()
        self.dbfile = dbfile

        @self.app.hook('before_request')
        def before_request():
            bottle.local.db = self.connect()

        @self.app.hook('after_request')
        def after_request():
            bottle.local.db.close()

    def connect(self):
        return sqlite3.connect(self.dbfile)

init_sqlite = SQlitePlugin # Alias for consistency
```

Now we can access the `connect()` method outside of a route callback and even reconfigure the plugin at runtime:

```
# [...] same as wiki-app example above
# but this time, we save the return value of init_sqlite()
sqlite_plugin = init_sqlite(dbfile='wiki.db')
```

```
if __name__ == '__main__':
    if 'development' in sys.argv:
        sqlite_plugin.dbfile = ':memory:' # reconfigure plugin
        db = sqlite_plugin.connect()       # reuse plugin methods
        db.execfile('test_database.sql')
        db.commit()
        db.close()
    run() # Run default app
```

Now if we call this script with a `development` command-line flag, it uses a memory-mapped test database instead of the real one.

CHAPTER

FIVE

LICENCE

Code and documentation are available according to the MIT Licence:

The Bottle logo however is *NOT* covered by that licence. 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.

PYTHON MODULE INDEX

INDEX