Bottle Documentation

Release 0.12.13

Marcel Hellkamp

July 21, 2018

Contents

1 User's Guide **3**
- 1.1 Tutorial . . . 3
- 1.2 Configuration (DRAFT) . . . 22
- 1.3 Request Routing . . . 25
- 1.4 SimpleTemplate Engine . . . 27
- 1.5 API Reference . . . 31
- 1.6 List of available Plugins . . . 44

2 Knowledge Base **49**
- 2.1 Tutorial: Todo-List Application . . . 49
- 2.2 Primer to Asynchronous Applications . . . 63
- 2.3 Recipes . . . 66
- 2.4 Frequently Asked Questions . . . 70

3 Development and Contribution **73**
- 3.1 Release Notes and Changelog . . . 73
- 3.2 Contributors . . . 76
- 3.3 Developer Notes . . . 78
- 3.4 Plugin Development Guide . . . 82

4 License **87**

Python Module Index **89**

Index **91**

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

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

Example: "Hello World" in a bottle

```
from bottle import route, run, template

@route('/hello/<name>')
def index(name):
    return template('<b>Hello {{name}}</b>!', name=name)

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

Run this script or paste it into a Python console, then point your browser to http://localhost:8080/hello/world. That's it.

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**.

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

CHAPTER 1

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.

Tutorial

This tutorial introduces you to the concepts and features of the Bottle web framework and covers basic and advanced topics alike. You can read it from start to end, or use it as a reference later on. The automatically generated API Reference may be interesting for you, too. It covers more details, but explains less than this tutorial. Solutions for the most common questions can be found in our Recipes collection or on the Frequently Asked Questions page. If you need any help, join our mailing list or visit us in our IRC channel.

Installation

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

```
$ wget http://bottlepy.org/bottle.py
```

This will get you the latest development snapshot that includes all the new features. If you prefer a more stable environment, you should stick with the stable releases. These are available on PyPI and can be installed via **pip** (recommended), **easy_install** or your package manager:

```
$ sudo pip install bottle              # recommended
$ sudo easy_install bottle             # alternative without pip
$ sudo apt-get install python-bottle   # works for debian, ubuntu, ...
```

Either way, you'll need Python 2.5 or newer (including 3.x) to run bottle applications. If you do not have permissions to install packages system-wide or simply don't want to, create a virtualenv first:

```
$ virtualenv develop              # Create virtual environment
$ source develop/bin/activate     # Change default python to virtual one
(develop)$ pip install -U bottle  # Install bottle to virtual environment
```

Or, if virtualenv is not installed on your system:

```
$ wget https://raw.github.com/pypa/virtualenv/master/virtualenv.py
$ python virtualenv.py develop    # Create virtual environment
$ source develop/bin/activate     # Change default python to virtual one
(develop)$ pip install -U bottle  # Install bottle to virtual environment
```

Quickstart: "Hello World"

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

```
from bottle import route, run

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

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

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

The *route()* decorator binds a piece of code to an URL path. In this case, we link the `/hello` path to the `hello()` function. This is called a *route* (hence the decorator name) and is the most important concept of this framework. You can define as many routes as you want. Whenever a browser requests an URL, the associated function is called and the return value is sent back to the browser. Its as simple as that.

The *run()* call in the last line starts a built-in development server. It runs on `localhost` port `8080` and serves requests until you hit `Control-c`. You can switch the server backend later, but for now a development server is all we need. It requires no setup at all and is an incredibly painless way to get your application up and running for local tests.

The *Debug Mode* is very helpful during early development, but should be switched off for public applications. Keep that in mind.

Of course this is a very simple example, but it shows the basic concept of how applications are built with Bottle. Continue reading and you'll see what else is possible.

The Default Application

For the sake of simplicity, most examples in this tutorial use a module-level *route()* decorator to define routes. This adds routes to a global "default application", an instance of *Bottle* that is automatically created the first time you call *route()*. Several other module-level decorators and functions relate to this default application, but if you prefer a more object oriented approach and don't mind the extra typing, you can create a separate application object and use that instead of the global one:

```
from bottle import Bottle, run

app = Bottle()

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

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

The object-oriented approach is further described in the *Default Application* section. Just keep in mind that you have a choice.

Request Routing

In the last chapter we built a very simple web application with only a single route. Here is the routing part of the "Hello World" example again:

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

The *route()* decorator links an URL path to a callback function, and adds a new route to the *default application*. An application with just one route is kind of boring, though. Let's add some more:

```
@route('/')
@route('/hello/<name>')
def greet(name='Stranger'):
    return template('Hello {{name}}, how are you?', name=name)
```

This example demonstrates two things: You can bind more than one route to a single callback, and you can add wildcards to URLs and access them via keyword arguments.

Dynamic Routes

Routes that contain wildcards are called *dynamic routes* (as opposed to *static routes*) and match more than one URL at the same time. A simple wildcard consists of a name enclosed in angle brackets (e.g. `<name>`) and accepts one or more characters up to the next slash (`/`). For example, the route `/hello/<name>` accepts requests for `/hello/alice` as well as `/hello/bob`, but not for `/hello`, `/hello/` or `/hello/mr/smith`.

Each wildcard passes the covered part of the URL as a keyword argument to the request callback. You can use them right away and implement RESTful, nice-looking and meaningful URLs with ease. Here are some other examples along with the URLs they'd match:

```
@route('/wiki/<pagename>')              # matches /wiki/Learning_Python
def show_wiki_page(pagename):
    ...

@route('/<action>/<user>')              # matches /follow/defnull
def user_api(action, user):
    ...
```

New in version 0.10.

Filters are used to define more specific wildcards, and/or transform the covered part of the URL before it is passed to the callback. A filtered wildcard is declared as `<name:filter>` or `<name:filter:config>`. The syntax for the optional config part depends on the filter used.

The following filters are implemented by default and more may be added:

- **:int** matches (signed) digits only and converts the value to integer.
- **:float** similar to :int but for decimal numbers.
- **:path** matches all characters including the slash character in a non-greedy way and can be used to match more than one path segment.
- **:re** allows you to specify a custom regular expression in the config field. The matched value is not modified.

Let's have a look at some practical examples:

```
@route('/object/<id:int>')
def callback(id):
    assert isinstance(id, int)

@route('/show/<name:re:[a-z]+>')
def callback(name):
    assert name.isalpha()
```

```
@route('/static/<path:path>')
def callback(path):
    return static_file(path, ...)
```

You can add your own filters as well. See `Routing` for details.

Changed in version 0.10.

The new rule syntax was introduced in **Bottle 0.10** to simplify some common use cases, but the old syntax still works and you can find a lot of code examples still using it. The differences are best described by example:

Old Syntax	New Syntax
`:name`	`<name>`
`:name#regexp#`	`<name:re:regexp>`
`:#regexp#`	`<:re:regexp>`
`:##`	`<:re>`

Try to avoid the old syntax in future projects if you can. It is not currently deprecated, but will be eventually.

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, 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 # or route

@get('/login') # or @route('/login')
def login():
    return '''
        <form action="/login" method="post">
            Username: <input name="username" type="text" />
            Password: <input name="password" type="password" />
            <input value="Login" type="submit" />
        </form>
    '''

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

In this example the `/login` URL is linked 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 *Request Data* section.

Special Methods: HEAD and ANY

The 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, change the wildcard to use the *path* filter:

```
@route('/static/<filepath:path>')
def server_static(filepath):
    return static_file(filepath, 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 plain error page. You can override the default for a specific HTTP status code with the *`error()`* decorator:

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

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

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

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 the `Content-Length` header set to 0.

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

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

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

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

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

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

Changing the Default Encoding

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

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

@route('/latin9')
def get_latin():
```

```
    response.content_type = 'text/html; charset=latin9'
    return u'ISO-8859-15 is also known as latin9.'
```

In some rare cases the Python encoding names differ from the names supported by the HTTP specification. Then, you have to do both: first set the `Response.content_type` header (which is sent to the client unchanged) and then set the `Response.charset` attribute (which is used to encode unicode).

Static Files

You can directly return file objects, but `static_file()` is the recommended way to serve static files. It automatically guesses a mime-type, adds a `Last-Modified` header, restricts paths to a `root` directory for security reasons and generates appropriate error responses (403 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 MIME type to disable guessing.

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

@route('/static/<filename:path>')
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:path>')
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 behavior to handle exceptions in your middleware by setting `bottle.app().catchall` to `False`.

The `Response` Object

Response metadata such as the HTTP status code, response headers and cookies are stored in an object called *response* up to the point where they are transmitted to the browser. You can manipulate these metadata 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 behavior 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 codes other than the ones defined by the HTTP specification will only confuse the browser and break standards.

Response Header

Response headers such as `Cache-Control` or `Location` are defined via `Response.set_header()`. This method takes two parameters, a header name and a value. The name part is case-insensitive:

```
@route('/wiki/<page>')
def wiki(page):
    response.set_header('Content-Language', 'en')
    ...
```

Most headers are unique, meaning that only one header per name is send to the client. Some special headers however are allowed to appear more than once in a response. To add an additional header, use `Response.add_header()` instead of `Response.set_header()`:

```
response.set_header('Set-Cookie', 'name=value')
response.add_header('Set-Cookie', 'name2=value2')
```

Please note that this is just an example. If you want to work with cookies, read *ahead*.

Cookies

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

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

The `Response.set_cookie()` method accepts a number of additional keyword arguments that control the cookies lifetime and behavior. Some of the most common settings are described here:

- **max_age:** Maximum age in seconds. (default: `None`)
- **expires:** A datetime object or UNIX timestamp. (default: `None`)
- **domain:** The domain that is allowed to read the cookie. (default: current domain)
- **path:** Limit the cookie to a given path (default: `/`)
- **secure:** Limit the cookie to HTTPS connections (default: off).
- **httponly:** Prevent client-side javascript to read this cookie (default: off, requires Python 2.6 or newer).

If neither *expires* nor *max_age* is set, the cookie expires at the end of the browser session or as soon as the browser window is closed. There are some other gotchas you should consider when using cookies:

- Cookies are limited to 4 KB of text in most browsers.
- Some users configure their browsers to not accept cookies at all. Most search engines ignore cookies too. Make sure that your application still works without cookies.
- Cookies are stored at client side and are not encrypted in any way. Whatever you store in a cookie, the user can read it. Worse than that, an attacker might be able to steal a user's cookies through XSS vulnerabilities on your side. Some viruses are known to read the browser cookies, too. Thus, never store confidential information in cookies.
- Cookies are easily forged by malicious clients. Do not trust cookies.

Signed Cookies

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

```
@route('/login')
def do_login():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if check_login(username, password):
        response.set_cookie("account", username, secret='some-secret-key')
        return template("<p>Welcome {{name}}! You are now logged in.</p>", name=username)
    else:
        return "<p>Login failed.</p>"

@route('/restricted')
def restricted_area():
    username = request.get_cookie("account", secret='some-secret-key')
    if username:
        return template("Hello {{name}}. Welcome back.", name=username)
```

```
    else:
        return "You are not logged in. Access denied."
```

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

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

Request Data

Cookies, HTTP header, HTML `<form>` fields and other request data is available through the global *request* object. This special object always refers to the *current* request, even in multi-threaded environments where multiple client connections are handled at the same time:

```
from bottle import request, route, template

@route('/hello')
def hello():
    name = request.cookies.username or 'Guest'
    return template('Hello {{name}}', name=name)
```

The *request* object is a subclass of *BaseRequest* and has a very rich API to access data. We only cover the most commonly used features here, but it should be enough to get started.

Introducing `FormsDict`

Bottle uses a special type of dictionary to store form data and cookies. *FormsDict* behaves like a normal dictionary, but has some additional features to make your life easier.

Attribute access: All values in the dictionary are also accessible as attributes. These virtual attributes return unicode strings, even if the value is missing or unicode decoding fails. In that case, the string is empty, but still present:

```
name = request.cookies.name

# is a shortcut for:

name = request.cookies.getunicode('name') # encoding='utf-8' (default)

# which basically does this:

try:
    name = request.cookies.get('name', '').decode('utf-8')
except UnicodeError:
    name = u''
```

Multiple values per key: *FormsDict* is a subclass of *MultiDict* and can store more than one value per key. The standard dictionary access methods will only return a single value, but the *getall()* method returns a (possibly empty) list of all values for a specific key:

```
for choice in request.forms.getall('multiple_choice'):
    do_something(choice)
```

WTForms support: Some libraries (e.g. WTForms) want all-unicode dictionaries as input. *FormsDict.decode()* does that for you. It decodes all values and returns a copy of itself, while preserving multiple values per key and all the other features.

Note: In **Python 2** all keys and values are byte-strings. If you need unicode, you can call *FormsDict.getunicode()* or fetch values via attribute access. Both methods try to decode the string (default: utf8) and return an empty string if that fails. No need to catch `UnicodeError`:

```
>>> request.query['city']
'G\xc3\xb6ttingen'  # A utf8 byte string
>>> request.query.city
u'Göttingen'        # The same string as unicode
```

In **Python 3** all strings are unicode, but HTTP is a byte-based wire protocol. The server has to decode the byte strings somehow before they are passed to the application. To be on the safe side, WSGI suggests ISO-8859-1 (aka latin1), a reversible single-byte codec that can be re-encoded with a different encoding later. Bottle does that for *FormsDict.getunicode()* and attribute access, but not for the dict-access methods. These return the unchanged values as provided by the server implementation, which is probably not what you want.

```
>>> request.query['city']
'GÃ¶ttingen' # An utf8 string provisionally decoded as ISO-8859-1 by the server
>>> request.query.city
'Göttingen'  # The same string correctly re-encoded as utf8 by bottle
```

If you need the whole dictionary with correctly decoded values (e.g. for WTForms), you can call *FormsDict.decode()* to get a re-encoded copy.

Cookies

Cookies are small pieces of text stored in the clients browser and sent back to the server with each request. They are useful to keep some state around for more than one request (HTTP itself is stateless), but should not be used for security related stuff. They can be easily forged by the client.

All cookies sent by the client are available through *BaseRequest.cookies* (a *FormsDict*). 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
```

The *BaseRequest.get_cookie()* method is a different way do access cookies. It supports decoding *signed cookies* as described in a separate section.

HTTP Headers

All HTTP headers sent by the client (e.g. `Referer`, `Agent` or `Accept-Language`) are stored in a *WSGIHeaderDict* and accessible through the *BaseRequest.headers* attribute. A *WSGIHeaderDict* is basically a dictionary with case-insensitive keys:

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

Query Variables

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 *BaseRequest.query* attribute (a *FormsDict*) to access these values and the *BaseRequest.query_string* attribute to get the whole string.

```
from bottle import route, request, response, template
@route('/forum')
def display_forum():
    forum_id = request.query.id
    page = request.query.page or '1'
    return template('Forum ID: {{id}} (page {{page}})', id=forum_id, page=page)
```

HTML *<form>* Handling

Let us start from the beginning. In HTML, a typical `<form>` looks something like this:

```
<form action="/login" method="post">
    Username: <input name="username" type="text" />
    Password: <input name="password" type="password" />
    <input value="Login" type="submit" />
</form>
```

The `action` attribute specifies the URL that will receive the form data. `method` defines the HTTP method to use (`GET` or `POST`). With `method="get"` the form values are appended to the URL and available through *BaseRequest.query* as described above. This is considered insecure and has other limitations, so we use `method="post"` here. If in doubt, use `POST` forms.

Form fields transmitted via `POST` are stored in *BaseRequest.forms* as a *FormsDict*. The server side code may look like this:

```
from bottle import route, request

@route('/login')
def login():
    return '''
        <form action="/login" method="post">
            Username: <input name="username" type="text" />
            Password: <input name="password" type="password" />
            <input value="Login" type="submit" />
        </form>
    '''

@route('/login', method='POST')
def do_login():
    username = request.forms.get('username')
    password = request.forms.get('password')
    if check_login(username, password):
```

```
        return "<p>Your login information was correct.</p>"
    else:
        return "<p>Login failed.</p>"
```

There are several other attributes used to access form data. Some of them combine values from different sources for easier access. The following table should give you a decent overview.

Attribute	GET Form fields	POST Form fields	File Uploads
BaseRequest.query	yes	no	no
BaseRequest.forms	no	yes	no
BaseRequest.files	no	no	yes
BaseRequest.params	yes	yes	no
BaseRequest.GET	yes	no	no
BaseRequest.POST	no	yes	yes

File uploads

To support file uploads, we have to change the `<form>` tag a bit. First, we tell the browser to encode the form data in a different way by adding an `enctype="multipart/form-data"` attribute to the `<form>` tag. Then, we add `<input type="file" />` tags to allow the user to select a file. Here is an example:

```
<form action="/upload" method="post" enctype="multipart/form-data">
  Category:      <input type="text" name="category" />
  Select a file: <input type="file" name="upload" />
  <input type="submit" value="Start upload" />
</form>
```

Bottle stores file uploads in *BaseRequest.files* as *FileUpload* instances, along with some metadata about the upload. Let us assume you just want to save the file to disk:

```
@route('/upload', method='POST')
def do_upload():
    category   = request.forms.get('category')
    upload     = request.files.get('upload')
    name, ext = os.path.splitext(upload.filename)
    if ext not in ('.png','.jpg','.jpeg'):
        return 'File extension not allowed.'

    save_path = get_save_path_for_category(category)
    upload.save(save_path) # appends upload.filename automatically
    return 'OK'
```

FileUpload.filename contains the name of the file on the clients file system, but is cleaned up and normalized to prevent bugs caused by unsupported characters or path segments in the filename. If you need the unmodified name as sent by the client, have a look at *FileUpload.raw_filename*.

The *FileUpload.save* method is highly recommended if you want to store the file to disk. It prevents some common errors (e.g. it does not overwrite existing files unless you tell it to) and stores the file in a memory efficient way. You can access the file object directly via *FileUpload.file*. Just be careful.

JSON Content

Some JavaScript or REST clients send `application/json` content to the server. The *BaseRequest.json* attribute contains the parsed data structure, if available.

The raw request body

You can access the raw body data as a file-like object via *BaseRequest.body*. This is a BytesIO buffer or a temporary file depending on the content length and *BaseRequest.MEMFILE_MAX* setting. In both cases the body is completely buffered before you can access the attribute. If you expect huge amounts of data and want to get direct unbuffered access to the stream, have a look at request['wsgi.input'].

WSGI Environment

Each *BaseRequest* instance wraps a WSGI environment dictionary. The original is stored in *BaseRequest.environ*, but the request object itself behaves like a dictionary, too. Most of the interesting data is exposed through special methods or attributes, but if you want to access WSGI environ variables directly, you can do so:

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

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. Its 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.

Plugins

New in version 0.9.

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

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

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

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

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

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

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

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

Application-wide Installation

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

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

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

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

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

Uninstall Plugins

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

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

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

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

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

Route-specific Installation

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

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

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

Blacklisting Plugins

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

```
sqlite_plugin = SQLitePlugin(dbfile='/tmp/test1.db')
install(sqlite_plugin)

dbfile1 = '/tmp/test1.db'
dbfile2 = '/tmp/test2.db'

@route('/open/<db>', skip=[sqlite_plugin])
```

```
def open_db(db):
    # The 'db' keyword argument is not touched by the plugin this time.

    # The plugin handle can be used for runtime configuration, too.
    if db == 'test1':
        sqlite_plugin.dbfile = dbfile1
    elif db == 'test2':
        sqlite_plugin.dbfile = dbfile2
    else:
        abort(404, "No such database.")

    return "Database File switched to: " + sqlite_plugin.dbfile
```

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

Plugins and Sub-Applications

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

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

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

root.install(plugins.WTForms())
```

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

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

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

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

Development

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

Default Application

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

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

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

```
app = Bottle()

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

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

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

```
default_app.push()

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

app = default_app.pop()
```

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

```
default_app.push()

import some.module

app = default_app.pop()
```

Debug Mode

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

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

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

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

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

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

Auto Reloading

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

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

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

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

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

Command Line Interface

Starting with version 0.10 you can use bottle as a command-line tool:

```
$ python -m bottle

Usage: bottle.py [options] package.module:app

Options:
  -h, --help            show this help message and exit
  --version             show version number.
  -b ADDRESS, --bind=ADDRESS
                        bind socket to ADDRESS.
  -s SERVER, --server=SERVER
                        use SERVER as backend.
  -p PLUGIN, --plugin=PLUGIN
                        install additional plugin/s.
  --debug               start server in debug mode.
  --reload              auto-reload on file changes.
```

The *ADDRESS* field takes an IP address or an IP:PORT pair and defaults to `localhost:8080`. The other parameters should be self-explanatory.

Both plugins and applications are specified via import expressions. These consist of an import path (e.g. `package.module`) and an expression to be evaluated in the namespace of that module, separated by a colon. See *load()* for details. Here are some examples:

```
# Grab the 'app' object from the 'myapp.controller' module and
# start a paste server on port 80 on all interfaces.
python -m bottle -server paste -bind 0.0.0.0:80 myapp.controller:app

# Start a self-reloading development server and serve the global
# default application. The routes are defined in 'test.py'
python -m bottle --debug --reload test

# Install a custom debug plugin with some parameters
python -m bottle --debug --reload --plugin 'utils:DebugPlugin(exc=True)'' test
```

```
# Serve an application that is created with 'myapp.controller.make_app()'
# on demand.
python -m bottle 'myapp.controller:make_app()''
```

Deployment

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

The easiest way to increase performance is to install a multi-threaded server library like paste or cherrypy and tell Bottle to use that instead of the single-threaded server:

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

This, and many other deployment options are described in a separate article: `deployment`

Glossary

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

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

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

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

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

Configuration (DRAFT)

Bottle applications can store their configuration in *`Bottle.config`*, a dict-like object and central place for application specific settings. This dictionary controls many aspects of the framework, tells (newer) plugins what to do, and can be used to store your own configuration as well.

Configuration Basics

The *`Bottle.config`* object behaves a lot like an ordinary dictionary. All the common dict methods work as expected. Let us start with some examples:

```
import bottle
app = bottle.default_app()              # or bottle.Bottle() if you prefer

app.config['autojson']    = False       # Turns off the "autojson" feature
app.config['sqlite.db']   = ':memory:'  # Tells the sqlite plugin which db to use
app.config['myapp.param'] = 'value'     # Example for a custom config value.

# Change many values at once
```

```
app.config.update({
    'autojson': False,
    'sqlite.db': ':memory:',
    'myapp.param': 'value'
})

# Add default values
app.config.setdefault('myapp.param2', 'some default')

# Receive values
param  = app.config['myapp.param']
param2 = app.config.get('myapp.param2', 'fallback value')

# An example route using configuration values
@app.route('/about', view='about.rst')
def about():
    email = app.config.get('my.email', 'nomail@example.com')
    return {'email': email}
```

The app object is not always available, but as long as you are within a request context, you can use the *request* object to get the current application and its configuration:

```
from bottle import request
def is_admin(user):
    return user == request.app.config['myapp.admin_user']
```

Naming Convention

To make life easier, plugins and applications should follow some simple rules when it comes to config parameter names:

- All keys should be lowercase strings and follow the rules for python identifiers (no special characters but the underscore).
- Namespaces are separated by dots (e.g. `namespace.field` or `namespace.subnamespace.field`).
- Bottle uses the root namespace for its own configuration. Plugins should store all their variables in their own namespace (e.g. `sqlite.db` or `werkzeug.use_debugger`).
- Your own application should use a separate namespace (e.g. `myapp.*`).

Loading Configuration from a File

Configuration files are useful if you want to enable non-programmers to configure your application, or just don't want to hack python module files just to change the database port. A very common syntax for configuration files is shown here:

```
[bottle]
debug = True

[sqlite]
db = /tmp/test.db
commit = auto

[myapp]
admin_user = defnull
```

With *ConfigDict.load_config()* you can load these `*.ini` style configuration files from disk and import their values into your existing configuration:

```
app.config.load_config('/etc/myapp.conf')
```

Loading Configuration from a nested `dict`

Another useful method is *ConfigDict.load_dict()*. This method takes an entire structure of nested dictionaries and turns it into a flat list of keys and values with namespaced keys:

```
# Load an entire dict structure
app.config.load_dict({
    'autojson': False,
    'sqlite': { 'db': ':memory:' },
    'myapp': {
        'param': 'value',
        'param2': 'value2'
    }
})

assert app.config['myapp.param'] == 'value'

# Load configuration from a json file
with open('/etc/myapp.json') as fp:
    app.config.load_dict(json.load(fp))
```

Listening to configuration changes

The `config` hook on the application object is triggered each time a value in *Bottle.config* is changed. This hook can be used to react on configuration changes at runtime, for example reconnect to a new database, change the debug settings on a background service or resize worker thread pools. The hook callback receives two arguments (key, new_value) and is called before the value is actually changed in the dictionary. Raising an exception from a hook callback cancels the change and the old value is preserved.

```
@app.hook('config')
def on_config_change(key, value):
  if key == 'debug':
      switch_own_debug_mode_to(value)
```

The hook callbacks cannot *change* the value that is to be stored to the dictionary. That is what filters are for.

Filters and other Meta Data

ConfigDict allows you to store meta data along with configuration keys. Two meta fields are currently defined:

help A help or description string. May be used by debugging, introspection or admin tools to help the site maintainer configuring their application.

filter A callable that accepts and returns a single value. If a filter is defined for a key, any new value stored to that key is first passed through the filter callback. The filter can be used to cast the value to a different type, check for invalid values (throw a ValueError) or trigger side effects.

This feature is most useful for plugins. They can validate their config parameters or trigger side effects using filters and document their configuration via `help` fields:

```
class SomePlugin(object):
    def setup(app):
        app.config.meta_set('some.int', 'filter', int)
        app.config.meta_set('some.list', 'filter',
            lambda val: str(val).split(';'))
        app.config.meta_set('some.list', 'help',
            'A semicolon separated list.')

    def apply(self, callback, route):
        ...

import bottle
app = bottle.default_app()
app.install(SomePlugin())

app.config['some.list'] = 'a;b;c'      # Actually stores ['a', 'b', 'c']
app.config['some.int'] = 'not an int' # raises ValueError
```

API Documentation

class ConfigDict (**a*, ***ka*)

A dict-like configuration storage with additional support for namespaces, validators, meta-data, on_change listeners and more.

This storage is optimized for fast read access. Retrieving a key or using non-altering dict methods (e.g. *dict.get()*) has no overhead compared to a native dict.

load_config (*filename*)

Load values from an *.ini style config file.

If the config file contains sections, their names are used as namespaces for the values within. The two special sections `DEFAULT` and `bottle` refer to the root namespace (no prefix).

load_dict (*source*, *namespace=''*, *make_namespaces=False*)

Import values from a dictionary structure. Nesting can be used to represent namespaces.

```
>>> ConfigDict().load_dict({'name': {'space': {'key': 'value'}}})
{'name.space.key': 'value'}
```

update (**a*, ***ka*)

If the first parameter is a string, all keys are prefixed with this namespace. Apart from that it works just as the usual dict.update(). Example: `update('some.namespace', key='value')`

meta_get (*key*, *metafield*, *default=None*)

Return the value of a meta field for a key.

meta_set (*key*, *metafield*, *value*)

Set the meta field for a key to a new value. This triggers the on-change handler for existing keys.

meta_list (*key*)

Return an iterable of meta field names defined for a key.

Request Routing

Bottle uses a powerful routing engine to find the right callback for each request. The *tutorial* shows you the basics. This document covers advanced techniques and rule mechanics in detail.

Rule Syntax

The `Router` distinguishes between two basic types of routes: **static routes** (e.g. `/contact`) and **dynamic routes** (e.g. `/hello/<name>`). A route that contains one or more *wildcards* it is considered dynamic. All other routes are static.

Changed in version 0.10.

The simplest form of a wildcard consists of a name enclosed in angle brackets (e.g. `<name>`). The name should be unique for a given route and form a valid python identifier (alphanumeric, starting with a letter). This is because wildcards are used as keyword arguments for the request callback later.

Each wildcard matches one or more characters, but stops at the first slash (`/`). This equals a regular expression of `[^/]+` and ensures that only one path segment is matched and routes with more than one wildcard stay unambiguous.

The rule `/<action>/<item>` matches as follows:

Path	Result
/save/123	`{'action': 'save', 'item': '123'}`
/save/123/	*No Match*
/save/	*No Match*
//123	*No Match*

You can change the exact behaviour in many ways using filters. This is described in the next section.

Wildcard Filters

New in version 0.10.

Filters are used to define more specific wildcards, and/or transform the matched part of the URL before it is passed to the callback. A filtered wildcard is declared as `<name:filter>` or `<name:filter:config>`. The syntax for the optional config part depends on the filter used.

The following standard filters are implemented:

- **:int** matches (signed) digits and converts the value to integer.
- **:float** similar to :int but for decimal numbers.
- **:path** matches all characters including the slash character in a non-greedy way and may be used to match more than one path segment.
- **:re[:exp]** allows you to specify a custom regular expression in the config field. The matched value is not modified.

You can add your own filters to the router. All you need is a function that returns three elements: A regular expression string, a callable to convert the URL fragment to a python value, and a callable that does the opposite. The filter function is called with the configuration string as the only parameter and may parse it as needed:

```
app = Bottle()

def list_filter(config):
    ''' Matches a comma separated list of numbers. '''
    delimiter = config or ','
    regexp = r'\d+(%s\d)*' % re.escape(delimiter)

    def to_python(match):
        return map(int, match.split(delimiter))

    def to_url(numbers):
        return delimiter.join(map(str, numbers))
```

```
    return regexp, to_python, to_url

app.router.add_filter('list', list_filter)

@app.route('/follow/<ids:list>')
def follow_users(ids):
    for id in ids:
        ...
```

Legacy Syntax

Changed in version 0.10.

The new rule syntax was introduce in **Bottle 0.10** to simplify some common use cases, but the old syntax still works and you can find lot code examples still using it. The differences are best described by example:

Old Syntax	New Syntax
`:name`	`<name>`
`:name#regexp#`	`<name:re:regexp>`
`:#regexp#`	`<:re:regexp>`
`:##`	`<:re>`

Try to avoid the old syntax in future projects if you can. It is not currently deprecated, but will be eventually.

Explicit routing configuration

Route decorator can also be directly called as method. This way provides flexibility in complex setups, allowing you to directly control, when and how routing configuration done.

Here is a basic example of explicit routing configuration for default bottle application:

```
def setup_routing():
    bottle.route('/', 'GET', index)
    bottle.route('/edit', ['GET', 'POST'], edit)
```

In fact, any `Bottle` instance routing can be configured same way:

```
def setup_routing(app):
    app.route('/new', ['GET', 'POST'], form_new)
    app.route('/edit', ['GET', 'POST'], form_edit)

app = Bottle()
setup_routing(app)
```

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.

`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 Expressions

You already learned the use of the `{{...}}` syntax from the "Hello World!" example above, but there is more: any python expression is allowed within the curly brackets as long as it evaluates to 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 expression 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 expression with an exclamation mark to disable escaping for that expression:

```
>>> 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 template engine allows you to embed lines or blocks of python code within your template. Code lines start with `%` and code blocks are surrounded by `<%` and `%>` tokens:

```
% name = "Bob"  # a line of python code
<p>Some plain text in between</p>
<%
  # A block of python code
  name = name.title().strip()
%>
<p>More plain text</p>
```

Embedded python code follows regular python syntax, but with two additional syntax rules:

- **Indentation is ignored.** You can put as much whitespace in front of statements as you want. This allows you to align your code with the surrounding markup and can greatly improve readability.
- Blocks that are normally indented now have to be closed explicitly with an `end` keyword.

```
<ul>
  % for item in basket:
    <li>{{item}}</li>
  % end
</ul>
```

Both the `%` and the `<%` tokens are only recognized if they are the first non-whitespace characters in a line. You don't have to escape them if they appear mid-text in your template markup. Only if a line of text starts with one of these tokens, you have to escape it with a backslash. In the rare case where the backslash + token combination appears in your markup at the beginning of a line, you can always help yourself with a string literal in an inline expression:

```
This line contains % and <% but no python code.
\% This text-line starts with the '%' token.
\<% Another line that starts with a token but is rendered as text.
{{'\\%'}} this line starts with an escaped token.
```

If you find yourself to escape a lot, consider using custom tokens.

Whitespace Control

Code blocks and code lines always span the whole line. Whitespace in front of after a code segment is stripped away. You won't see empty lines or dangling whitespace in your template because of embedded code:

```
<div>
 % if True:
  <span>content</span>
 % end
</div>
```

This snippet renders to clean and compact html:

```
<div>
  <span>content</span>
</div>
```

But embedding code still requires you to start a new line, which may not what you want to see in your rendered template. To skip the newline in front of a code segment, end the text line with a double-backslash:

```
<div>\\
 %if True:
<span>content</span>\\
 %end
</div>
```

THis time the rendered template looks like this:

```
<div><span>content</span></div>
```

This only works directly in front of code segments. In all other places you can control the whitespace yourself and don't need any special syntax.

Template Functions

Each template is preloaded with a bunch of functions that help with the most common use cases. These functions are always available. You don't have to import or provide them yourself. For everything not covered here there are probably good python libraries available. Remember that you can `import` anything you want within your templates. They are python programs after all.

Changed in version 0.12: Prior to this release, *include()* and *rebase()* were sytnax keywords, not functions.

include(*sub_template*, ***variables*)
Render a sub-template with the specified variables and insert the resulting text into the current template. The function returns a dictionary containing the local variables passed to or defined within the sub-template:

```
% include('header.tpl', title='Page Title')
Page Content
% include('footer.tpl')
```

rebase(*name*, ***variables*)
Mark the current template to be later included into a different template. After the current template is rendered, its resulting text is stored in a variable named `base` and passed to the base-template, which is then rendered. This can be used to *wrap* a template with surrounding text, or simulate the inheritance feature found in other template engines:

```
% rebase('base.tpl', title='Page Title')
<p>Page Content ...</p>
```

This can be combined with the following `base.tpl`:

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

Accessing undefined variables in a template raises `NameError` and stops rendering immediately. This is standard python behavior and nothing new, but vanilla python lacks an easy way to check the availability of a variable. This quickly gets annoying if you want to support flexible inputs or use the same template in different situations. These functions may help:

defined(*name*)
Return True if the variable is defined in the current template namespace, False otherwise.

get(*name*, *default=None*)
Return the variable, or a default value.

setdefault(*name*, *default*)
If the variable is not defined, create it with the given default value. Return the variable.

Here is an example that uses all three functions to implement optional template variables in different ways:

```
% setdefault('text', 'No Text')
<h1>{{get('title', 'No Title')}}</h1>
<p> {{ text }} </p>
% if defined('author'):
  <p>By {{ author }}</p>
% end
```

SimpleTemplate API

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

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

API Reference

This is a mostly auto-generated API. If you are new to bottle, you might find the narrative Tutorial more helpful.

Module Contents

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

debug(*mode=True*)
Change the debug level. There is only one debug level supported at the moment.

run(*app=None*, *server='wsgiref'*, *host='127.0.0.1'*, *port=8080*, *interval=1*, *reloader=False*, *quiet=False*, *plugins=None*, *debug=None*, ***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)
- **port** – Server port to bind to. Values below 1024 require root privileges. (default: 8080)
- **reloader** – Start auto-reloading server? (default: False)
- **interval** – Auto-reloader interval in seconds (default: 1)
- **quiet** – Suppress output to stdout and stderr? (default: False)
- **options** – Options passed to the server adapter.

load(*target*, ***namespace*)
Import a module or fetch an object from a module.

- `package.module` returns *module* as a module object.
- `pack.mod:name` returns the module variable *name* from *pack.mod*.
- `pack.mod:func()` calls *pack.mod.func()* and returns the result.

The last form accepts not only function calls, but any type of expression. Keyword arguments passed to this function are available as local variables. Example: `import_string('re:compile(x)', x='[a-z]')`

load_app(*target*)
Load a bottle application from a module and make sure that the import does not affect the current default application, but returns a separate application object. See *load()* for the target parameter.

request = <LocalRequest: GET http://127.0.0.1/>
A thread-safe instance of *LocalRequest*. If accessed from within a request callback, this instance always refers to the *current* request (even on a multithreaded server).

response = Content-Type: text/html; charset=UTF-8
A thread-safe instance of *LocalResponse*. It is used to change the HTTP response for the *current* request.

HTTP_CODES = {300: 'Multiple Choices', 301: 'Moved Permanently', 302: 'Found', 303: 'See Other', 304: 'Not Modified', 305:
A dict to map HTTP status codes (e.g. 404) to phrases (e.g. 'Not Found')

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

Routing

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

route(*path*, *method='GET'*, *callback=None*, ***options*)
get(...)
post(...)
put(...)
delete(...)
Decorator to install a route to the current default application. See *Bottle.route()* for details.

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

WSGI and HTTP Utilities

parse_date(*ims*)
Parse rfc1123, rfc850 and asctime timestamps and return UTC epoch.

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

cookie_encode(*data*, *key*)
Encode and sign a pickle-able object. Return a (byte) string

cookie_decode(*data*, *key*)
Verify and decode an encoded string. Return an object or None.

cookie_is_encoded(*data*)
Return True if the argument looks like a encoded cookie.

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

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

path_shift(*script_name*, *path_info*, *shift=1*)
Shift path fragments from PATH_INFO to SCRIPT_NAME and vice versa.

Returns The modified paths.

Parameters

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

Data Structures

class MultiDict(**a*, ***k*)
This dict stores multiple values per key, but behaves exactly like a normal dict in that it returns only the newest value for any given key. There are special methods available to access the full list of values.

get(*key*, *default=None*, *index=-1*, *type=None*)
Return the most recent value for a key.

Parameters

- **default** – The default value to be returned if the key is not present or the type conversion fails.
- **index** – An index for the list of available values.
- **type** – If defined, this callable is used to cast the value into a specific type. Exception are suppressed and result in the default value to be returned.

append(*key*, *value*)
Add a new value to the list of values for this key.

replace(*key*, *value*)
Replace the list of values with a single value.

getall(*key*)
Return a (possibly empty) list of values for a key.

getone(*key*, *default=None*, *index=-1*, *type=None*)
Aliases for WTForms to mimic other multi-dict APIs (Django)

getlist(*key*)
Return a (possibly empty) list of values for a key.

class HeaderDict(**a*, ***ka*)
A case-insensitive version of *MultiDict* that defaults to replace the old value instead of appending it.

class FormsDict (**a*, ***k*)
This *MultiDict* subclass is used to store request form data. Additionally to the normal dict-like item access methods (which return unmodified data as native strings), this container also supports attribute-like access to its values. Attributes are automatically de- or recoded to match *input_encoding* (default: 'utf8'). Missing attributes default to an empty string.

input_encoding = 'utf8'
Encoding used for attribute values.

recode_unicode = True
If true (default), unicode strings are first encoded with *latin1* and then decoded to match *input_encoding*.

decode (*encoding=None*)
Returns a copy with all keys and values de- or recoded to match *input_encoding*. Some libraries (e.g. WTForms) want a unicode dictionary.

getunicode (*name*, *default=None*, *encoding=None*)
Return the value as a unicode string, or the default.

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

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

cgikeys = ('CONTENT_TYPE', 'CONTENT_LENGTH')
List of keys that do not have a `HTTP_` prefix.

raw (*key*, *default=None*)
Return the header value as is (may be bytes or unicode).

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

pop ()
Return the current default application and remove it from the stack.

push (*value=None*)
Add a new *Bottle* instance to the stack

class ResourceManager (*base='./'*, *opener=<built-in function open>*, *cachemode='all'*)
This class manages a list of search paths and helps to find and open application-bound resources (files).

Parameters

- **base** – default value for *add_path()* calls.
- **opener** – callable used to open resources.
- **cachemode** – controls which lookups are cached. One of 'all', 'found' or 'none'.

path = None
A list of search paths. See *add_path()* for details.

cache = None
A cache for resolved paths. `res.cache.clear()` clears the cache.

add_path (*path*, *base=None*, *index=None*, *create=False*)
Add a new path to the list of search paths. Return False if the path does not exist.

Parameters

- **path** – The new search path. Relative paths are turned into an absolute and normalized form. If the path looks like a file (not ending in /), the filename is stripped off.
- **base** – Path used to absolutize relative search paths. Defaults to `base` which defaults to `os.getcwd()`.
- **index** – Position within the list of search paths. Defaults to last index (appends to the list).

The *base* parameter makes it easy to reference files installed along with a python module or package:

```
res.add_path('./resources/', __file__)
```

lookup(*name*)
Search for a resource and return an absolute file path, or *None*.

The *path* list is searched in order. The first match is returend. Symlinks are followed. The result is cached to speed up future lookups.

open(*name*, *mode='r'*, **args*, ***kwargs*)
Find a resource and return a file object, or raise IOError.

class FileUpload(*fileobj*, *name*, *filename*, *headers=None*)

file = None
Open file(-like) object (BytesIO buffer or temporary file)

name = None
Name of the upload form field

raw_filename = None
Raw filename as sent by the client (may contain unsafe characters)

headers = None
A *HeaderDict* with additional headers (e.g. content-type)

content_type
Current value of the 'Content-Type' header.

content_length
Current value of the 'Content-Length' header.

get_header(*name*, *default=None*)
Return the value of a header within the mulripart part.

filename
Name of the file on the client file system, but normalized to ensure file system compatibility. An empty filename is returned as 'empty'.

Only ASCII letters, digits, dashes, underscores and dots are allowed in the final filename. Accents are removed, if possible. Whitespace is replaced by a single dash. Leading or tailing dots or dashes are removed. The filename is limited to 255 characters.

save(*destination*, *overwrite=False*, *chunk_size=65536*)
Save file to disk or copy its content to an open file(-like) object. If *destination* is a directory, *filename* is added to the path. Existing files are not overwritten by default (IOError).

Parameters

- **destination** – File path, directory or file(-like) object.
- **overwrite** – If True, replace existing files. (default: False)

- **chunk_size** – Bytes to read at a time. (default: 64kb)

Exceptions

exception BottleException
A base class for exceptions used by bottle.

The Bottle Class

class Bottle(*catchall=True*, *autojson=True*)
Each Bottle object represents a single, distinct web application and consists of routes, callbacks, plugins, resources and configuration. Instances are callable WSGI applications.

Parameters **catchall** – If true (default), handle all exceptions. Turn off to let debugging middleware handle exceptions.

config = None
A *ConfigDict* for app specific configuration.

resources = None
A *ResourceManager* for application files

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

add_hook(*name*, *func*)
Attach a callback to a hook. Three hooks are currently implemented:

before_request Executed once before each request. The request context is available, but no routing has happened yet.

after_request Executed once after each request regardless of its outcome.

app_reset Called whenever *Bottle.reset()* is called.

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

trigger_hook(*_Bottle__name*, **args*, ***kwargs*)
Trigger a hook and return a list of results.

hook(*name*)
Return a decorator that attaches a callback to a hook. See *add_hook()* for details.

mount(*prefix*, *app*, ***options*)
Mount an application (*Bottle* or plain WSGI) to a specific URL prefix. Example:

```
root_app.mount('/admin/', admin_app)
```

Parameters

- **prefix** – path prefix or *mount-point*. If it ends in a slash, that slash is mandatory.
- **app** – an instance of *Bottle* or a WSGI application.

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

merge(*routes*)
Merge the routes of another *Bottle* application or a list of *Route* objects into this application. The routes keep their 'owner', meaning that the *Route.app* attribute is not changed.

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

uninstall(*plugin*)
Uninstall plugins. Pass an instance to remove a specific plugin, a type object to remove all plugins that match that type, a string to remove all plugins with a matching `name` attribute or `True` to remove all plugins. Return the list of removed plugins.

reset(*route=None*)
Reset all routes (force plugins to be re-applied) and clear all caches. If an ID or route object is given, only that specific route is affected.

close()
Close the application and all installed plugins.

run(***kwargs*)
Calls *run()* with the same parameters.

match(*environ*)
Search for a matching route and return a (*Route* , urlargs) tuple. The second value is a dictionary with parameters extracted from the URL. Raise *HTTPError* (404/405) on a non-match.

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

add_route(*route*)
Add a route object, but do not change the *Route.app* attribute.

route(*path=None*, *method='GET'*, *callback=None*, *name=None*, *apply=None*, *skip=None*, ***config*)
A decorator to bind a function to a request URL. Example:

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

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

Parameters

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

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

get(*path=None*, *method='GET'*, ***options*)
Equals *route()*.

post(*path=None*, *method='POST'*, ***options*)
Equals *route()* with a POST method parameter.

put(*path=None*, *method='PUT'*, ***options*)
Equals *route()* with a PUT method parameter.

delete(*path=None*, *method='DELETE'*, ***options*)
Equals *route()* with a DELETE method parameter.

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

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

class **Route**(*app*, *rule*, *method*, *callback*, *name=None*, *plugins=None*, *skiplist=None*, ***config*)
This class wraps a route callback along with route specific metadata and configuration and applies Plugins on demand. It is also responsible for turing an URL path rule into a regular expression usable by the Router.

app = None
The application this route is installed to.

rule = None
The path-rule string (e.g. /wiki/:page).

method = None
The HTTP method as a string (e.g. GET).

callback = None
The original callback with no plugins applied. Useful for introspection.

name = None
The name of the route (if specified) or None.

plugins = None
A list of route-specific plugins (see *Bottle.route()*).

skiplist = None
A list of plugins to not apply to this route (see *Bottle.route()*).

config = None
Additional keyword arguments passed to the *Bottle.route()* decorator are stored in this dictionary. Used for route-specific plugin configuration and meta-data.

call
The route callback with all plugins applied. This property is created on demand and then cached to speed up subsequent requests.

reset()
Forget any cached values. The next time *call* is accessed, all plugins are re-applied.

prepare()
Do all on-demand work immediately (useful for debugging).

all_plugins()
Yield all Plugins affecting this route.

get_undecorated_callback()
Return the callback. If the callback is a decorated function, try to recover the original function.

get_callback_args()
Return a list of argument names the callback (most likely) accepts as keyword arguments. If the callback is a decorated function, try to recover the original function before inspection.

get_config(*key*, *default=None*)
Lookup a config field and return its value, first checking the route.config, then route.app.config.

The `Request` Object

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.

Request
alias of *BaseRequest*

class BaseRequest(*environ=None*)
A wrapper for WSGI environment dictionaries that adds a lot of convenient access methods and properties. Most of them are read-only.

Adding new attributes to a request actually adds them to the environ dictionary (as 'bottle.request.ext.<name>'). This is the recommended way to store and access request-specific data.

MEMFILE_MAX = 102400
Maximum size of memory buffer for *body* in bytes.

environ
The wrapped WSGI environ dictionary. This is the only real attribute. All other attributes actually are read-only properties.

app
Bottle application handling this request.

route
The bottle *Route* object that matches this request.

url_args
The arguments extracted from the URL.

path
The value of `PATH_INFO` with exactly one prefixed slash (to fix broken clients and avoid the "empty path" edge case).

method
The `REQUEST_METHOD` value as an uppercase string.

headers
A *WSGIHeaderDict* that provides case-insensitive access to HTTP request headers.

get_header(*name*, *default=None*)
Return the value of a request header, or a given default value.

cookies
Cookies parsed into a *FormsDict*. Signed cookies are NOT decoded. Use *get_cookie()* if you expect signed cookies.

get_cookie(*key*, *default=None*, *secret=None*)
Return the content of a cookie. To read a *Signed Cookie*, the *secret* must match the one used to create the cookie (see *BaseResponse.set_cookie()*). If anything goes wrong (missing cookie or wrong signature), return a default value.

query
The *query_string* parsed into a *FormsDict*. These values are sometimes called "URL arguments" or "GET parameters", but not to be confused with "URL wildcards" as they are provided by the `Router`.

forms
Form values parsed from an *url-encoded* or *multipart/form-data* encoded POST or PUT request body. The result is returned as a *FormsDict*. All keys and values are strings. File uploads are stored separately in *files*.

params
A *FormsDict* with the combined values of *query* and *forms*. File uploads are stored in *files*.

files
File uploads parsed from *multipart/form-data* encoded POST or PUT request body. The values are instances of *FileUpload*.

json
If the `Content-Type` header is `application/json`, this property holds the parsed content of the request body. Only requests smaller than *MEMFILE_MAX* are processed to avoid memory exhaustion.

body
The HTTP request body as a seek-able file-like object. Depending on *MEMFILE_MAX*, this is either a temporary file or a `io.BytesIO` instance. Accessing this property for the first time reads and replaces the `wsgi.input` environ variable. Subsequent accesses just do a *seek(0)* on the file object.

chunked
True if Chunked transfer encoding was.

GET
An alias for *query*.

POST
The values of *forms* and *files* combined into a single *FormsDict*. Values are either strings (form values) or instances of `cgi.FieldStorage` (file uploads).

url
The full request URI including hostname and scheme. If your app lives behind a reverse proxy or load balancer and you get confusing results, make sure that the `X-Forwarded-Host` header is set correctly.

urlparts
The *url* string as an `urlparse.SplitResult` tuple. The tuple contains (scheme, host, path, query_string and fragment), but the fragment is always empty because it is not visible to the server.

fullpath
Request path including *script_name* (if present).

query_string
The raw *query* part of the URL (everything in between `?` and `#`) as a string.

script_name
The initial portion of the URL's *path* that was removed by a higher level (server or routing middleware) before the application was called. This script path is returned with leading and tailing slashes.

path_shift(*shift=1*)

Shift path segments from *path* to *script_name* and vice versa.

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

content_length
The request body length as an integer. The client is responsible to set this header. Otherwise, the real length of the body is unknown and -1 is returned. In this case, *body* will be empty.

content_type
The Content-Type header as a lowercase-string (default: empty).

is_xhr
True if the request was triggered by a XMLHttpRequest. This only works with JavaScript libraries that support the *X-Requested-With* header (most of the popular libraries do).

is_ajax
Alias for *is_xhr*. "Ajax" is not the right term.

auth
HTTP authentication data as a (user, password) tuple. This implementation currently supports basic (not digest) authentication only. If the authentication happened at a higher level (e.g. in the front web-server or a middleware), the password field is None, but the user field is looked up from the `REMOTE_USER` environ variable. On any errors, None is returned.

remote_route
A list of all IPs that were involved in this request, starting with the client IP and followed by zero or more proxies. This does only work if all proxies support the `'X-Forwarded-For` header. Note that this information can be forged by malicious clients.

remote_addr
The client IP as a string. Note that this information can be forged by malicious clients.

copy()
Return a new *Request* with a shallow *environ* copy.

The module-level *bottle.request* is a proxy object (implemented in *LocalRequest*) and always refers to the *current* request, or in other words, the request that is currently processed by the request handler in the current thread. This *thread locality* ensures that you can safely use a global instance in a multi-threaded environment.

class LocalRequest(*environ=None*)
A thread-local subclass of *BaseRequest* with a different set of attributes for each thread. There is usually only one global instance of this class (*request*). If accessed during a request/response cycle, this instance always refers to the *current* request (even on a multithreaded server).

bind(*environ=None*)
Wrap a WSGI environ dictionary.

environ
Thread-local property

request = <LocalRequest: GET http://127.0.0.1/>
A thread-safe instance of *LocalRequest*. If accessed from within a request callback, this instance always refers to the *current* request (even on a multithreaded server).

The Response Object

The *Response* class stores the HTTP status code as well as headers and cookies that are to be sent to the client. Similar to *bottle.request* there is a thread-local *bottle.response* instance that can be used to adjust the *current* response. Moreover, you can instantiate *Response* and return it from your request handler. In this case, the custom instance overrules the headers and cookies defined in the global one.

Response
alias of *BaseResponse*

class BaseResponse(*body=''*, *status=None*, *headers=None*, ***more_headers*)
Storage class for a response body as well as headers and cookies.

This class does support dict-like case-insensitive item-access to headers, but is NOT a dict. Most notably, iterating over a response yields parts of the body and not the headers.

Parameters

- **body** – The response body as one of the supported types.
- **status** – Either an HTTP status code (e.g. 200) or a status line including the reason phrase (e.g. '200 OK').
- **headers** – A dictionary or a list of name-value pairs.

Additional keyword arguments are added to the list of headers. Underscores in the header name are replaced with dashes.

copy(*cls=None*)
Returns a copy of self.

status_line
The HTTP status line as a string (e.g. `404 Not Found`).

status_code
The HTTP status code as an integer (e.g. 404).

status
A writeable property to change the HTTP response status. It accepts either a numeric code (100-999) or a string with a custom reason phrase (e.g. "404 Brain not found"). Both *status_line* and *status_code* are updated accordingly. The return value is always a status string.

headers
An instance of *HeaderDict*, a case-insensitive dict-like view on the response headers.

get_header(*name*, *default=None*)
Return the value of a previously defined header. If there is no header with that name, return a default value.

set_header(*name*, *value*)
Create a new response header, replacing any previously defined headers with the same name.

add_header(*name*, *value*)
Add an additional response header, not removing duplicates.

iter_headers()
Yield (header, value) tuples, skipping headers that are not allowed with the current response status code.

headerlist
WSGI conform list of (header, value) tuples.

content_type
Current value of the 'Content-Type' header.

content_length
Current value of the 'Content-Length' header.

expires
Current value of the 'Expires' header.

charset
Return the charset specified in the content-type header (default: utf8).

set_cookie(*name*, *value*, *secret=None*, ***options*)
Create a new cookie or replace an old one. If the *secret* parameter is set, create a *Signed Cookie* (described below).

Parameters

- **name** – the name of the cookie.
- **value** – the value of the cookie.
- **secret** – a signature key required for signed cookies.

Additionally, this method accepts all RFC 2109 attributes that are supported by `cookie.Morsel`, including:

Parameters

- **max_age** – maximum age in seconds. (default: None)
- **expires** – a datetime object or UNIX timestamp. (default: None)
- **domain** – the domain that is allowed to read the cookie. (default: current domain)
- **path** – limits the cookie to a given path (default: current path)
- **secure** – limit the cookie to HTTPS connections (default: off).
- **httponly** – prevents client-side javascript to read this cookie (default: off, requires Python 2.6 or newer).

If neither *expires* nor *max_age* is set (default), the cookie will expire at the end of the browser session (as soon as the browser window is closed).

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

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

delete_cookie(*key*, ***kwargs*)
Delete a cookie. Be sure to use the same *domain* and *path* settings as used to create the cookie.

class LocalResponse(*body=''*, *status=None*, *headers=None*, ***more_headers*)
A thread-local subclass of *BaseResponse* with a different set of attributes for each thread. There is usually only one global instance of this class (*response*). Its attributes are used to build the HTTP response at the end of the request/response cycle.

body
Thread-local property

The following two classes can be raised as an exception. The most noticeable difference is that bottle invokes error handlers for *HTTPError*, but not for *HTTPResponse* or other response types.

exception HTTPResponse(*body=''*, *status=None*, *headers=None*, ***more_headers*)

exception HTTPError(*status=None*, *body=None*, *exception=None*, *traceback=None*, ***options*)

Templates

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

class BaseTemplate(*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, ***settings*)
Base class and minimal API for template adapters

__init__(*source=None*, *name=None*, *lookup=[]*, *encoding='utf8'*, ***settings*)
Create a new template. If the source parameter (str or buffer) is missing, the name argument is used to guess a template filename. Subclasses can assume that self.source and/or self.filename are set. Both are strings. The lookup, encoding and settings parameters are stored as instance variables. The lookup

parameter stores a list containing directory paths. The encoding parameter should be used to decode byte strings or files. The settings parameter contains a dict for engine-specific settings.

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

classmethod global_config(*key*, **args*)
This reads or sets the global settings stored in class.settings.

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

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

view(*tpl_name*, ***defaults*)
Decorator: renders a template for a handler. The handler can control its behavior like that:

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

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

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

Class	URL	Decorator	Render function
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
```

List of available Plugins

This is a list of third-party plugins that add extend Bottles core functionality or integrate other libraries with the Bottle framework.

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

Bottle-Cork Cork provides a simple set of methods to implement Authentication and Authorization in web applications based on Bottle.

Bottle-Extras Meta package to install the bottle plugin collection.

Bottle-Flash flash plugin for bottle

Bottle-Hotqueue FIFO Queue for Bottle built upon redis

Macaron Macaron is an object-relational mapper (ORM) for SQLite.

Bottle-Memcache Memcache integration for Bottle.

Bottle-MongoDB MongoDB integration for Bottle

Bottle-Redis Redis integration for Bottle.

Bottle-Renderer Renderer plugin for bottle

Bottle-Servefiles A reusable app that serves static files for bottle apps

Bottle-Sqlalchemy SQLAlchemy integration for Bottle.

Bottle-Sqlite SQLite3 database integration for Bottle.

Bottle-Web2pydal Web2py Dal integration for Bottle.

Bottle-Werkzeug Integrates the *werkzeug* library (alternative request and response objects, advanced debugging middleware and more).

Plugins listed here are not part of Bottle or the Bottle project, but developed and maintained by third parties.

Bottle-SQLite

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

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

Installation

Install with one of the following commands:

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

or download the latest version from github:

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

Usage

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

```
import bottle

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

@app.route('/show/:item')
```

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

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

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

Configuration

The following configuration options exist for the plugin class:

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

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

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

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

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

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

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

Bottle-Werkzeug

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

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

Installation

Install with one of the following commands:

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

or download the latest version from github:

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

Usage

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

```
import bottle
from bottle.ext import werkzeug

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

req = werkzeug.request # For the lazy.

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

Using the Debugger

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

Configuration

The following configuration options exist for the plugin class:

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

CHAPTER 2

Knowledge Base

A collection of articles, guides and HOWTOs.

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

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

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

Before We Start...

Install Bottle

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

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

Further Software Necessities

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

Create An SQL Database

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

```
import sqlite3
con = sqlite3.connect('todo.db') # Warning: This file is created in the current directory
con.execute("CREATE TABLE todo (id INTEGER PRIMARY KEY, task char(100) NOT NULL, status bool NOT NULL
con.execute("INSERT INTO todo (task,status) VALUES ('Read A-byte-of-python to get a good 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)")
con.commit()
```

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).

Using Bottle for a Web-Based ToDo List

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

Understanding routes

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

First Step - Showing All Open Items

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

```
import sqlite3
from bottle import route, run

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

run()
```

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

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

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

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

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

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

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

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

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

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

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

Debugging and Auto-Reload

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

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

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

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

Another quite 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 developing, not on production 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 col in row:
    <td>{{col}}</td>
  %end
  </tr>
%end
</table>
```

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

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

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

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

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

Using GET and POST Values

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

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

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

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

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

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

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

    conn.commit()
    c.close()

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

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

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

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

The code needs to be extended to:

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

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

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

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

        conn.commit()
        c.close()

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

`new_task.tpl` looks like this:

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

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

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

Now we are able to extend our to do list.

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

Editing Existing Items

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

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

The basic statement for a dynamic route looks like this:

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

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

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

The code looks like this:

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

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

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

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

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

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

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

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

```
%#template for editing a task
%#the template expects to receive a value for "no" as well a "old", the text of the selected ToDo 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, as demonstrated later.

Validating Dynamic Routes

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

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

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

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

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

Dynamic Routes Using Regular Expressions

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

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

As said above, the solution is a regular expression:

```
@route('/item:item#[0-9]+#')
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_#[0-9]+#)` 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, static_file

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

At first, we need to import the `static_file` function from Bottle. As you can see, the `return static_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 `static_file`, which would be here `mimetype='text/html'`. `static_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#[0-9]+#')
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]}
```

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.

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 settings of the standard server first.

Running Bottle on a different port and IP

As standard, Bottle serves 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 Paste server, use the following code:

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

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

Running Bottle on Apache with mod_wsgi

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

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

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`

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 tasks and serve 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 .

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, static_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#[0-9]+#')
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():

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

@route('/json:json#[0-9]+#')
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 col in row:
    <td>{{col}}</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" maxlength="100" name="task">
<input type="submit" name="save" value="save">
</form>
```

Primer to Asynchronous Applications

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

The Limits of Synchronous WSGI

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

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

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

Greenlets to the rescue

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

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

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

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

from time import sleep
from bottle import route, run

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

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

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

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

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

Event Callbacks

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

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

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

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

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

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

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

Finally: WebSockets

Lets forget about the low-level details for a while and speak about WebSockets. Since you are reading this article, you probably know what WebSockets are: A bidirectional communication channel between a browser (client) and a web application (server).

Thankfully the gevent-websocket package does all the hard work for us. Here is a simple WebSocket endpoint that receives messages and just sends them back to the client:

```
from bottle import request, Bottle, abort
app = Bottle()

@app.route('/websocket')
def handle_websocket():
    wsock = request.environ.get('wsgi.websocket')
    if not wsock:
        abort(400, 'Expected WebSocket request.')

    while True:
        try:
            message = wsock.receive()
            wsock.send("Your message was: %r" % message)
        except WebSocketError:
            break

from gevent.pywsgi import WSGIServer
from geventwebsocket import WebSocketHandler, WebSocketError
server = WSGIServer(("0.0.0.0", 8080), app,
                    handler_class=WebSocketHandler)
server.serve_forever()
```

The while-loop runs until the client closes the connection. You get the idea :)

The client-site JavaScript API is really straight forward, too:

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript">
    var ws = new WebSocket("ws://example.com:8080/websocket");
    ws.onopen = function() {
        ws.send("Hello, world");
    };
    ws.onmessage = function (evt) {
        alert(evt.data);
    };
  </script>
</head>
</html>
```

Recipes

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

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)
```

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 powerful 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**.

Unit-Testing Bottle Applications

Unit-testing is usually performed against methods defined in your web application without running a WSGI environment.

A simple example using Nose:

```
import bottle

@bottle.route('/')
def index():
    return 'Hi!'

if __name__ == '__main__':
    bottle.run()
```

Test script:

```
import mywebapp

def test_webapp_index():
    assert mywebapp.index() == 'Hi!'
```

In the example the Bottle route() method is never executed - only index() is tested.

Functional Testing Bottle Applications

Any HTTP-based testing system can be used with a running WSGI server, but some testing frameworks work more intimately with WSGI, and provide the ability the call WSGI applications in a controlled environment, with tracebacks and full use of debugging tools. Testing tools for WSGI is a good starting point.

Example using WebTest and Nose:

```
from webtest import TestApp
import mywebapp

def test_functional_login_logout():
    app = TestApp(mywebapp.app)

    app.post('/login', {'user': 'foo', 'pass': 'bar'}) # log in and get a cookie

    assert app.get('/admin').status == '200 OK'        # fetch a page successfully

    app.get('/logout')                                 # log out
    app.reset()                                        # drop the cookie
```

```
    # fetch the same page, unsuccessfully
    assert app.get('/admin').status == '401 Unauthorized'
```

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='ANY')
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.

Ignore trailing slashes

For Bottle, `/example` and `/example/` are two different routes [1]. 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=myapp)
```

Keep-alive requests

Note: For a more detailed explanation, see Primer to Asynchronous Applications.

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

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

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

import time
from bottle import route, run

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

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

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

Gzip Compression in Bottle

Note: For a detailed discussion, see compression

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

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

- Compress on the fly and be fast doing so.
- Do not compress for browsers that don't support it.
- Do not compress files that are compressed already (images, videos).
- Do not compress dynamic files.
- Support two differed compression algorithms (gzip and deflate).
- Cache compressed files that don't change often.
- De-validate the cache if one of the files changed anyway.
- Make sure the cache does not get to big.
- Do not cache small files because a disk seek would take longer than on-the-fly compression.

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

Using the hooks plugin

For example, if you want to allow Cross-Origin Resource Sharing for the content returned by all of your URL, you can use the hook decorator and setup a callback function:

```
from bottle import hook, response, route

@hook('after_request')
def enable_cors():
    response.headers['Access-Control-Allow-Origin'] = '*'

@route('/foo')
def say_foo():
    return 'foo!'

@route('/bar')
def say_bar():
    return {'type': 'friendly', 'content': 'Hi!'}
```

You can also use the `before_request` to take an action before every function gets called.

Using Bottle with Heroku

Heroku, a popular cloud application platform now provides support for running Python applications on their infastructure.

This recipe is based upon the Heroku Quickstart, with Bottle specific code replacing the Write Your App section of the Getting Started with Python on Heroku/Cedar guide:

```
import os
from bottle import route, run

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

run(host="0.0.0.0", port=int(os.environ.get("PORT", 5000)))
```

Heroku's app stack passes the port that the application needs to listen on for requests, using the *os.environ* dictionary.

Frequently Asked Questions

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.

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.

Problems with reverse proxies

Redirects and url-building only works if bottle knows the public address and location of your application. If you run bottle locally behind a reverse proxy or load balancer, some information might get lost along the way. For example, the `wsgi.url_scheme` value or the `Host` header might reflect the local request by your proxy, not the real request by the client. Here is a small WSGI middleware snippet that helps to fix these values:

```
def fix_environ_middleware(app):
  def fixed_app(environ, start_response):
    environ['wsgi.url_scheme'] = 'https'
    environ['HTTP_X_FORWARDED_HOST'] = 'example.com'
    return app(environ, start_response)
  return https_app

app = bottle.default_app()
app.wsgi = fix_environ_middleware(app.wsgi)
```

CHAPTER 3

Development and Contribution

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

Release Notes and Changelog

Release 0.12

- New SimpleTemplate parser implementation * Support for multi-line code blocks (<% ... %>). * The keywords *include* and *rebase* are functions now and can accept variable template names.
- The new *BaseRequest.route()* property returns the *Route* that originally matched the request.
- Removed the BaseRequest.MAX_PARAMS limit. The hash collision bug in CPythons dict() implementation was fixed over a year ago. If you are still using Python 2.5 in production, consider upgrading or at least make sure that you get security fixed from your distributor.
- New *ConfigDict* API (see Configuration (DRAFT))

More information can be found in this development blog post.

Release 0.11

- Native support for Python 2.x and 3.x syntax. No need to run 2to3 anymore.
- Support for partial downloads (Range header) in static_file().
- The new *ResourceManager* interface helps locating files bundled with an application.
- Added a server adapter for waitress.
- New *Bottle.merge()* method to install all routes from one application into another.
- New *BaseRequest.app* property to get the application object that handles a request.
- Added *FormsDict.decode()* to get an all-unicode version (needed by WTForms).
- *MultiDict* and subclasses are now pickle-able.

API Changes

- Response.status is a read-write property that can be assigned either a numeric status code or a status string with a reason phrase (200 OK). The return value is now a string to better match existing APIs (WebOb,

werkzeug). To be absolutely clear, you can use the read-only properties *BaseResponse.status_code* and *BaseResponse.status_line*.

API Deprecations

- `SimpleTALTemplate` is now deprecating. There seems to be no demand.

Release 0.10

- Plugin API v2
  - To use the new API, set *Plugin.api* to `2`.
  - *Plugin.apply()* receives a *Route* object instead of a context dictionary as second parameter. The new object offers some additional information and may be extended in the future.
  - Plugin names are considered unique now. The topmost plugin with a given name on a given route is installed, all other plugins with the same name are silently ignored.
- The Request/Response Objects
  - Added *BaseRequest.json*, *BaseRequest.remote_route*, *BaseRequest.remote_addr*, *BaseRequest.query* and *BaseRequest.script_name*.
  - Added *BaseResponse.status_line* and *BaseResponse.status_code* attributes. In future releases, *BaseResponse.status* will return a string (e.g. `200 OK`) instead of an integer to match the API of other common frameworks. To make the transition as smooth as possible, you should use the verbose attributes from now on.
  - Replaced *MultiDict* with a specialized *FormsDict* in many places. The new dict implementation allows attribute access and handles unicode form values transparently.
- Templates
  - Added three new functions to the SimpleTemplate default namespace that handle undefined variables: *stpl.defined()*, *stpl.get()* and *stpl.setdefault()*.
  - The default escape function for SimpleTemplate now additionally escapes single and double quotes.
- Routing
  - A new route syntax (e.g. `/object/<id:int>`) and support for route wildcard filters.
  - Four new wildcard filters: *int*, *float*, *path* and *re*.
- Other changes
  - Added command line interface to load applications and start servers.
  - Introduced a *ConfigDict* that makes accessing configuration a lot easier (attribute access and auto-expanding namespaces).
  - Added support for raw WSGI applications to *Bottle.mount()*.
  - *Bottle.mount()* parameter order changed.
  - *Bottle.route()* now accpets an import string for the `callback` parameter.
  - Dropped Gunicorn 0.8 support. Current supported version is 0.13.
  - Added custom options to Gunicorn server.
  - Finally dropped support for type filters. Replace with a custom plugin of needed.

Release 0.9

Whats new?

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

Performance improvements

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

API changes

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

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

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.

Contributors

Bottle is written and maintained by Marcel Hellkamp <marc@bottlepy.org>.

Thanks to all the people who found bugs, sent patches, spread the word, helped each other on the mailing-list and made this project possible. I hope the following (alphabetically sorted) list is complete. If you miss your name on that list (or want your name removed) please `tell me` or add it yourself.

- acasajus
- Adam R. Smith
- Alexey Borzenkov
- Alexis Daboville
- Anton I. Sipos
- Anton Kolechkin
- apexi200sx

- apheage
- BillMa
- Brad Greenlee
- Brandon Gilmore
- Branko Vukelic
- Brian Sierakowski
- Brian Wickman
- Carl Scharenberg
- Damien Degois
- David Buxton
- Duane Johnson
- fcamel
- Frank Murphy
- Frederic Junod
- goldfaber3012
- Greg Milby
- gstein
- Ian Davis
- Itamar Nabriski
- Iuri de Silvio
- Jaimie Murdock
- Jeff Nichols
- Jeremy Kelley
- joegester
- Johannes Krampf
- Jonas Haag
- Joshua Roesslein
- Karl
- Kevin Zuber
- Kraken
- Kyle Fritz
- m35
- Marcos Neves
- masklinn
- Michael Labbe
- Michael Soulier

- reddit
- Nicolas Vanhoren
- Robert Rollins
- rogererens
- rwxrwx
- Santiago Gala
- Sean M. Collins
- Sebastian Wollrath
- Seth
- Sigurd Høgsbro
- Stuart Rackham
- Sun Ning
- Tomás A. Schertel
- Tristan Zajonc
- voltron
- Wieland Hoffmann
- zombat

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!

Get involved

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

- **Mailing list**: Join our mailing list by sending an email to bottlepy+subscribe@googlegroups.com (no google account required).
- **Twitter**: Follow us on Twitter or search for the #bottlepy tag.
- **IRC**: Join #bottlepy on irc.freenode.net or use the web chat interface.
- **Google plus**: We sometimes blog about Bottle, releases and technical stuff on our Google+ page.

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 or just download the latest development version as a source archive.

- **git:** `git clone git://github.com/defnull/bottle.git`
- **git/https:** `git clone https://github.com/defnull/bottle.git`

- **Download:** Development branch as tar archive or zip file.

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 (x.0) 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 (x.y) 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 about that.

Revision (x.y.z) 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.

Repository Structure

The source repository is structured as follows:

`master` branch 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` branches 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` branches 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 deleted 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. We'll do that on integration. You never know in which order we 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.

Submitting Patches

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.

Building the Documentation

You need a recent version of Sphinx to build the documentation. The recommended way is to install **`virtualenv`** using your distribution package repository and install sphinx manually to get an up-to-date version.

```
# Install prerequisites
which virtualenv || sudo apt-get install python-virtualenv
virtualenv --no-site-dependencies venv
./venv/pip install -U sphinx

# Clone or download bottle from github
git clone https://github.com/defnull/bottle.git

# Activate build environment
source ./venv/bin/activate

# Build HTML docs
cd bottle/docs
make html

# Optional: Install prerequisites for PDF generation
sudo apt-get install texlive-latex-extra \
                     texlive-latex-recommended \
                     texlive-fonts-recommended

# Optional: Build the documentation as PDF
make latex
```

```
cd ../build/docs/latex
make pdf
```

GIT Workflow Examples

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

First of all you need 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`.

Plugin Development Guide

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

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

How Plugins Work: The Basics

The plugin API builds on the concept of decorators. To put it briefly, a plugin is a decorator applied to every single route callback of an application.

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

```
from bottle import response, install
import time

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

install(stopwatch)
```

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

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

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

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

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

Plugin API

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

class Plugin (*object*)

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

name

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

api

The Plugin API is still evolving. This integer attribute tells bottle which version to use. If it is missing, bottle defaults to the first version. The current version is `2`. See *Plugin API changes* for details.

setup (*self*, *app*)

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

__call__ (*self*, *callback*)

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

apply (*self*, *callback*, *route*)

If defined, this method is used in favor of *__call__()* to decorate route callbacks. The additional *route* parameter is an instance of *Route* and provides a lot of meta-information and context for that route. See *The Route Context* for details.

close (*self*)

Called immediately before the plugin is uninstalled or the application is closed (see *Bottle.uninstall()* or *Bottle.close()*).

Both *Plugin.setup()* and *Plugin.close()* are *not* called for plugins that are applied directly to a route via the *Bottle.route()* decorator, but only for plugins installed to an application.

Plugin API changes

The Plugin API is still evolving and changed with Bottle 0.10 to address certain issues with the route context dictionary. To ensure backwards compatibility with 0.9 Plugins, we added an optional *Plugin.api* attribute to tell bottle which API to use. The API differences are summarized here.

- **Bottle 0.9 API 1** (*Plugin.api* not present)
  - Original Plugin API as described in the 0.9 docs.
- **Bottle 0.10 API 2** (*Plugin.api* equals 2)
  - The *context* parameter of the *Plugin.apply()* method is now an instance of *Route* instead of a context dictionary.

The Route Context

The *Route* instance passed to *Plugin.apply()* provides detailed informations about the associated route. The most important attributes are summarized here:

Attribute	Description
app	The application object this route is installed to.
rule	The rule string (e.g. `/wiki/:page`).
method	The HTTP method as a string (e.g. `GET`).
callback	The original callback with no plugins applied. Useful for introspection.
name	The name of the route (if specified) or `None`.
plugins	A list of route-specific plugins. These are applied in addition to application-wide plugins. (see *Bottle.route()*).
skiplist	A list of plugins to not apply to this route (again, see *Bottle.route()*).
config	Additional keyword arguments passed to the *Bottle.route()* decorator are stored in this dictionary. Used for route-specific configuration and meta-data.

For your plugin, *Route.config* is probably the most important attribute. Keep in mind that this dictionary is local to the route, but shared between all plugins. It is always a good idea to add a unique prefix or, if your plugin needs a lot of configuration, store it in a separate namespace within the *config* dictionary. This helps to avoid naming collisions between plugins.

Changing the `Route` object

While some *Route* attributes are mutable, changes may have unwanted effects on other plugins. It is most likely a bad idea to monkey-patch a broken route instead of providing a helpful error message and let the user fix the problem.

In some rare cases, however, it might be justifiable to break this rule. After you made your changes to the *Route* instance, raise `RouteReset` as an exception. This removes the current route from the cache and causes all plugins to be re-applied. The router is not updated, however. Changes to *rule* or *method* values have no effect on the router, but only on plugins. This may change in the future, though.

Runtime optimizations

Once all plugins are applied to a route, the wrapped route callback is cached to speed up subsequent requests. If the behavior of your plugin depends on configuration, and you want to be able to change that configuration at runtime, you need to read the configuration on each request. Easy enough.

For performance reasons, however, it might be worthwhile to choose a different wrapper based on current needs, work with closures, or enable or disable a plugin at runtime. Let's take the built-in HooksPlugin as an example: If no hooks are installed, the plugin removes itself from all affected routes and has virtaully no overhead. As soon as you install the first hook, the plugin activates itself and takes effect again.

To achieve this, you need control over the callback cache: *Route.reset()* clears the cache for a single route and *Bottle.reset()* clears all caches for all routes of an application at once. On the next request, all plugins are re-applied to the route as if it were requested for the first time.

Both methods won't affect the current request if called from within a route callback, of cause. To force a restart of the current request, raise `RouteReset` as an exception.

Plugin Example: SQLitePlugin

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

```
import sqlite3
import inspect

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

    name = 'sqlite'
    api = 2

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

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

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

```
        autocommit = conf.get('autocommit', self.autocommit)
        dictrows = conf.get('dictrows', self.dictrows)
        keyword = conf.get('keyword', self.keyword)

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

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

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

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

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

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

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

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

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

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

CHAPTER 4

License

Code and documentation are available according to the MIT License:

```
Copyright (c) 2012, Marcel Hellkamp.

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

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

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

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

b

bottle, 3

Symbols

__call__() (Plugin method), 83
__init__() (BaseTemplate method), 43

A

add_header() (BaseResponse method), 42
add_hook() (Bottle method), 36
add_path() (ResourceManager method), 34
add_route() (Bottle method), 37
all_plugins() (Route method), 38
api (Plugin attribute), 83
app (BaseRequest attribute), 39
app (Route attribute), 38
app() (in module bottle), 32
append() (MultiDict method), 33
apply() (Plugin method), 83
AppStack (class in bottle), 34
auth (BaseRequest attribute), 41

B

BaseRequest (class in bottle), 39
BaseResponse (class in bottle), 41
BaseTemplate (class in bottle), 43
bind() (LocalRequest method), 41
body (BaseRequest attribute), 40
body (LocalResponse attribute), 43
Bottle (class in bottle), 36
bottle (module), 3, 31, 44, 66, 70, 82
BottleException, 36

C

cache (ResourceManager attribute), 34
call (Route attribute), 38
callback, **22**
callback (Route attribute), 38
catchall (Bottle attribute), 36
cgikeys (WSGIHeaderDict attribute), 34
charset (BaseResponse attribute), 42
chunked (BaseRequest attribute), 40
close() (Bottle method), 37
close() (Plugin method), 83
config (Bottle attribute), 36
config (Route attribute), 38
ConfigDict (class in bottle), 25
content_length (BaseRequest attribute), 40
content_length (BaseResponse attribute), 42
content_length (FileUpload attribute), 35
content_type (BaseRequest attribute), 40
content_type (BaseResponse attribute), 42
content_type (FileUpload attribute), 35
cookie_decode() (in module bottle), 32
cookie_encode() (in module bottle), 32
cookie_is_encoded() (in module bottle), 32
cookies (BaseRequest attribute), 39
copy() (BaseRequest method), 41
copy() (BaseResponse method), 42

D

debug() (in module bottle), 31
decode() (FormsDict method), 34
decorator, **22**
default_app() (in module bottle), 32
defined() (in module stpl), 30
delete() (Bottle method), 38
delete() (in module bottle), 32
delete_cookie() (BaseResponse method), 43

E

environ, **22**
environ (BaseRequest attribute), 39
environ (LocalRequest attribute), 41
error() (Bottle method), 38
error() (in module bottle), 32
expires (BaseResponse attribute), 42

F

file (FileUpload attribute), 35
filename (FileUpload attribute), 35
files (BaseRequest attribute), 40
FileUpload (class in bottle), 35

forms (BaseRequest attribute), 39
FormsDict (class in bottle), 33
fullpath (BaseRequest attribute), 40

G

GET (BaseRequest attribute), 40
get() (Bottle method), 37
get() (in module bottle), 32
get() (in module stpl), 30
get() (MultiDict method), 33
get_callback_args() (Route method), 38
get_config() (Route method), 38
get_cookie() (BaseRequest method), 39
get_header() (BaseRequest method), 39
get_header() (BaseResponse method), 42
get_header() (FileUpload method), 35
get_undecorated_callback() (Route method), 38
get_url() (Bottle method), 37
getall() (MultiDict method), 33
getlist() (MultiDict method), 33
getone() (MultiDict method), 33
getunicode() (FormsDict method), 34
global_config() (bottle.BaseTemplate class method), 44

H

handler function, **22**
HeaderDict (class in bottle), 33
headerlist (BaseResponse attribute), 42
headers (BaseRequest attribute), 39
headers (BaseResponse attribute), 42
headers (FileUpload attribute), 35
hook() (Bottle method), 36
HTTP_CODES (in module bottle), 32
HTTPError, 43
HTTPResponse, 43

I

include() (in module stpl), 30
input_encoding (FormsDict attribute), 34
install() (Bottle method), 36
is_ajax (BaseRequest attribute), 41
is_xhr (BaseRequest attribute), 41
iter_headers() (BaseResponse method), 42

J

json (BaseRequest attribute), 40

L

load() (in module bottle), 31
load_app() (in module bottle), 32
load_config() (ConfigDict method), 25
load_dict() (ConfigDict method), 25
LocalRequest (class in bottle), 41
LocalResponse (class in bottle), 43
lookup() (ResourceManager method), 35

M

match() (Bottle method), 37
MEMFILE_MAX (BaseRequest attribute), 39
merge() (Bottle method), 36
meta_get() (ConfigDict method), 25
meta_list() (ConfigDict method), 25
meta_set() (ConfigDict method), 25
method (BaseRequest attribute), 39
method (Route attribute), 38
mount() (Bottle method), 36
MultiDict (class in bottle), 33

N

name (FileUpload attribute), 35
name (Plugin attribute), 83
name (Route attribute), 38

O

open() (ResourceManager method), 35

P

params (BaseRequest attribute), 40
parse_auth() (in module bottle), 32
parse_date() (in module bottle), 32
path (BaseRequest attribute), 39
path (ResourceManager attribute), 34
path_shift() (BaseRequest method), 40
path_shift() (in module bottle), 33
Plugin (class in bottle), 83
plugins (Route attribute), 38
pop() (AppStack method), 34
POST (BaseRequest attribute), 40
post() (Bottle method), 37
post() (in module bottle), 32
prepare() (BaseTemplate method), 44
prepare() (Route method), 38
push() (AppStack method), 34
put() (Bottle method), 38
put() (in module bottle), 32

Q

query (BaseRequest attribute), 39
query_string (BaseRequest attribute), 40

R

raw() (WSGIHeaderDict method), 34
raw_filename (FileUpload attribute), 35
rebase() (in module stpl), 30
recode_unicode (FormsDict attribute), 34
remote_addr (BaseRequest attribute), 41

remote_route (BaseRequest attribute), 41
remove_hook() (Bottle method), 36
render() (BaseTemplate method), 44
render() (SimpleTemplate method), 31
replace() (MultiDict method), 33
Request (in module bottle), 39
request (in module bottle), 32, 41
reset() (Bottle method), 37
reset() (Route method), 38
ResourceManager (class in bottle), 34
resources (Bottle attribute), 36
Response (in module bottle), 41
response (in module bottle), 32
route (BaseRequest attribute), 39
Route (class in bottle), 38
route() (Bottle method), 37
route() (in module bottle), 32
rule (Route attribute), 38
run() (Bottle method), 37
run() (in module bottle), 31

S

save() (FileUpload method), 35
script_name (BaseRequest attribute), 40
search() (bottle.BaseTemplate class method), 44
set_cookie() (BaseResponse method), 42
set_header() (BaseResponse method), 42
setdefault() (in module stpl), 31
setup() (Plugin method), 83
SimpleTemplate (class in bottle), 31
skiplist (Route attribute), 38
source directory, **22**
status (BaseResponse attribute), 42
status_code (BaseResponse attribute), 42
status_line (BaseResponse attribute), 42

T

template() (in module bottle), 44
trigger_hook() (Bottle method), 36

U

uninstall() (Bottle method), 37
update() (ConfigDict method), 25
url (BaseRequest attribute), 40
url_args (BaseRequest attribute), 39
urlparts (BaseRequest attribute), 40

V

view() (in module bottle), 44

W

wsgi() (Bottle method), 38
WSGIHeaderDict (class in bottle), 34

Y

yieldroutes() (in module bottle), 33