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 2.6 or newer (including 3.2+) 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 a URL, the associated function is called and the return value is sent back to the browser. It' s 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.

This is just a demonstration of 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 (don' t forget `from bottle import template`):

```
@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):
    ...
```

Filters can be 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 *Request Routing* for details.

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, DELETE or PATCH, add a `method` keyword argument to the `route()` decorator or use one of the five alternative decorators: `get()`, `post()`, `put()`, `delete()` or `patch()`.

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 beeing 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'

run()
```

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 default 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'

app.run()
```

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.

New in version 0.13.

Starting with bottle-0.13 you can use `Bottle` instances as context managers:

```
app = Bottle()

with app:

    # Our application object is now the default
    # for all shortcut functions and decorators

    assert my_app is default_app()

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

    # Also useful to capture routes defined in other modules
    import some_package.more_routes
```

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.
  -c FILE, --conf=FILE  load config values from FILE.
  -C NAME=VALUE, --param=NAME=VALUE
                        override config values.
  --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.