Fiole Documentation

Release 0.4.1

Florent Xicluna

July 03, 2014

Contents

Contents:

CHAPTER 1

Overview

`fiole.py` is a WSGI micro-framework with the following development constraints:

- Single file, **no external dependency**
- Provide enough features to build a web application with minimal effort
- Embed **a compact template engine**
- Keep the module reasonably small

Main features:

- *Routing*
- *Methods GET/HEAD/POST/PUT/DELETE*
- *Error handlers*
- File uploads (`Request.POST`)
- *Static files*
- *Fast template engine*
- Secure cookies (`Request.get_cookie()`, `Response.set_cookie()`, ...)

Disclaimer: this framework is intentionally limited. If you need a robust and scalable solution, look elsewhere.

Link to the PyPI page: https://pypi.python.org/pypi/fiole

Tested against: Python 2.7, PyPy 2.2 and Python >= 3.2

1.1 Quickstart

Either download the single file `fiole.py` and save it in your project directory, or `pip install fiole`, preferably in a `virtualenv`.

Create an application and save it with name `hello.py`:

```
from fiole import get, run_fiole


@get('/')
def index(request):
    return 'Hello World!'
```

```
run_fiole()
```

Then run this example (default port 8080) with:

```
python hello.py
```

or (on port 4000 for example):

```
python fiole.py -p 4000 hello
# (or)
python -m fiole -p 4000 hello
```

1.2 Next steps

Clone the examples and run the demo:

```
git clone git://github.com/florentx/fiole.git
cd fiole/
python fiole.py examples
```

Open your browser and navigate through the examples: http://127.0.0.1:8080

Read the documentation about *Routing requests* and *Build templates*.

Some features are not yet covered in the documentation:

- sending and receiving cookies
- adding custom HTTP headers
- stacking multiple applications
- serving through a third-party WSGI server (gevent, ...)

Look at the *Fiole API* for some crispy details.

Read the documentation of Flask and Bottle for more information about web development in general.

CHAPTER 2

Routing requests

2.1 Decorators

Declare the routes of your application using the Fiole decorators.

Example:

```
@route('/about')
def about(request):
    return "About Fiole sample application"

@get('/home')
def home(request):
    return "Sweet"
```

Both `route()` and `get()` declare a route for the `GET` (and `HEAD`) methods. The other decorators are `post()`, `put()` and `delete()`.

The `route()` decorator supports an extended syntax to match multiple methods to the same function:

```
@route('/api', methods=('GET', 'HEAD', 'POST'))
def multi(request):
    return "Received %s %s" % (request.method, request.path)
```

It is also available as a plain function:

```
def say_hi(request):
    return "Hi"

route('/welcome', methods=('GET', 'HEAD'), callback=say_hi)
route('/ciao', methods=('GET', 'HEAD'), callback=say_hi)
```

2.2 Dynamic routes

Two different syntaxes are supported.

Easy syntax

The simple syntax for URL pattern is like `"/hello/<name>"`. The placeholder matches a non-empty path element (excluding `"/"`):

```
@get('/')
@get('/hello/<name>')
def ciao(request, name='Stranger'):
    return render_template('Hello {{name}}!', name=name)
```

Regex syntax

The advanced syntax is regex-based. The variables are extracted from the named groups of the regular expression: `(?P<name>...)`. See Regular Expressions for the full syntax. The unnamed groups are ignored. The initial `^` and the final `$` chars should be omitted: they are automatically added when the route is registered.

The pattern parser switches to the advanced syntax when an extension notation is detected: `(?`.

Example:

```
@get(r'/api/(?P<action>[^:]+):?(?P<params>.*)')
def call_api(request, action, params):
    if action == 'view':
        return "Params received {0}".format(params)
    raise NotFound("This action is not supported")
```

2.3 Hooks

There's a flexible way to define extensions for `Fiole`: you can register hooks which will be executed for each request. For example you can setup a database connection before each request, and release the connection after the request.

A dumb no-op hook looks like:

```
app = get_app()

@app.hooks.append
def noop_hook(request):
    # acquire the resource
    # ...
    try:
        # pre-process the Request
        # ...
        response = yield
        # post-process the Response
        # ...
        yield response
    finally:
        # release the resource
        pass
```

This example setup a database connection:

```
@app.hooks.append
def hook_db(request):
    request.db = connect_db()
    try:
        # forward the response unchanged
        yield (yield)
    finally:
        request.db.close()
```

2.4 Helpers

Redirect a request:

```
@get('/test_redirect')
def test_redirect(request):
    raise Redirect('/hello')
```

Return an error page:

```
@get('/test_404')
def test_404(request):
    raise NotFound('Not here, sorry.')
```

Register a different error page:

```
template_404 = get_template("error404.tmpl")

@errorhandler(404)
def not_found(request):
    return template_404.render(request)
```

Send static files:

```
get_app().static_folder = "/path/to/public/directory"

@get('/static/(?P<path>.+)')
def download(request, path):
    return send_file(request, path)
```

CHAPTER 3

Build templates

Parts of this page are converted from the wheezy.template documentation. Thank you to the author. (*akorn*)

`Fiole` comes with a decent template engine. It supports the usual features of well known engines (*Mako*, *Jinja2*). The engine is derived from the wonderful wheezy.template package. It retains the same design goals:

- intuitive, use the full power of Python
- inherit your templates (`%extends`, `%include` and `%import`)
- stay *blazingly* fast.

In a nutshell, the syntax looks as simple as Bottle SimpleTemplate:

- `{{ ... }}` executes the enclosed *expression* and inserts the result.
- Use the `|e` filter to convert any of `& < > " '` to HTML-safe sequences.
- Switch to auto-escaping with `fiole.engine.default_filters = ['e']` and use `|n` to disable escaping (and default filters) for an expression.
- A single percent `%` at the beginning of the line identifies a *template directive* or *Python code*. (except if it is repeated: `%%`)
- Spaces and indentation around the `%` special char are ignored.
- A backslash `\` at the end of a line will skip the line ending.

Simple template:

```
%require(user, items)
Welcome, {{user.name}}!
%if items:
    %for i in items:
        {{i.name}}: {{i.price}}.
    %endfor
%else:
    No item found.
%endif
```

3.1 Template loading

This is a basic example for a route mapped to a template:

```
@get('/')
@get('/hello/<name>')
def hello(request, name=None):
    return render_template(
        source='Hello {{party.title() if party else "stranger"}}!', party=name)
```

In this case the template is not cached. It is built again for each request.

In order to activate the cache, the string template should be assigned to a name. The previous example becomes:

```
# Preload the cache
get_template('hello_tpl',
             source="""Hello {{party.title() if party else "stranger"}}!""",
             require=['party'])

@get('/')
@get('/hello/<name>')
def hello(request, name=None):
    return render_template('hello_tpl', party=name)
```

Templates can be saved to files in the `./templates/` folder of the project. Then they are loaded by filename and cached in memory:

```
@get('/')
def index(request):
    return render_template('hello.tmpl', party='World')
```

3.2 Inline expressions

3.2.1 Variables

The variables which need to be extracted from the context are listed in the `require` directive. These names become visible to the end of the template scope (a template is like a Python function). The application passes variables to the template via context:

```
%require(var1, var2)

{{ var1 }} ... {{ var2 }}
```

For string templates, you can declare the variables using the `require` keyword argument:

```
>>> hello_tmpl = get_template(source='Hello {{ party.capitalize() }}!', require=['party'])
>>> hello_tmpl.render(party='WORLD')
u'Hello World!'
>>> hello_tmpl.render({'party': 'world'})
u'Hello World!'
```

This declaration is omitted when rendering the string directly:

```
>>> render_template(source='Hello {{party}}!', party='World')
u'Hello World!'
>>> #
>>> render_template(source='Hello {{ party.capitalize() }}!', party='world')
u'Hello World!'
```

Variable syntax is not limited to a single name access. You are able to use the full power of Python to access items in dictionary, attributes, function calls, etc...

3.2.2 Filters

Variables can be formatted by filters. Filters are separated from the variable by the `|` symbol. Filter syntax:

```
{{ variable_name|filter1|filter2 }}
```

The filters are applied from left to right so above syntax is equivalent to the following call:

```
{{ filter2(filter1(variable_name)) }}
```

The built-in filter `|e` converts any of `& < > " '` to HTML-safe sequences `& < > &quot; &#x27;`.

You can define and use custom filters. Here is an example how to switch to a different implementation for the html escape filter:

```
try:
    from webext import escape_html
    engine.global_vars['escape'] = escape_html
except ImportError:
    pass
```

It tries to import an optimized version of html escape from the Webext package and assigns it to the `escape` global variable, which is aliased as `e` filter. The built-in `escape` is pure Python.

An example which demonstrates the standard `|e` filter:

```
>>> render_template('This: {{ data | e }}', data='"Hello small\' <i>World!<i>" ... & farther')
u'This: &quot;Hello small&#x27; &lt;i&gt;World!&lt;i&gt;&quot; ... &amp; farther'
```

You can enable auto-escaping by default, then use `|n` as the last filter to bypass the default filters:

```
>>> engine.default_filters = ['e']
>>> render_template(source='Hello {{ party.capitalize() }}',
...                 party='<script src="evil" />')
u'Hello &lt;script src=&quot;evil&quot; /&gt;'
```

```
>>> render_template(source='Hello {{ party | n }}',
...                 party='<em>World</em>')
u'Hello <em>World</em>'
```

You are able to use engine `Engine.global_vars` dictionary in order to simplify your template access to some commonly used variables.

3.3 Directives

Any line starting with a single `%` contains either a template directive or Python code. Following directives are supported:

- `%extends("layout.tmpl")`: Tell which master template should be extended to generate the current document. This should be the first line.
- `%require(firstname, lastname)`: Declare the variables which are expected when rendering the template.
- `%include("footer.html")`: Render the template and insert the output just here.
- `%import "widgets.tmpl" as widgets`: Import reusable helpers from another template.
- `%from "toolbox.tmpl" import popup`: Import a function from the other template.
- `%def`: Define a Python function (used for inheritance: `%extends` or `%import`).
- `%end` or `%enddef`: End the Python function definition.

3.3.1 Inheritance

Template inheritance (`%extends`) allows to build a master template that contains common layout of your site and defines areas that child templates can override.

Master Template

Master template is used to provide common layout of your site. Let define master template (filename `shared/master.html`):

```
<html>
  <head>
    <title>
    %def title():
    %enddef
    {{title()}} - My Site</title>
  </head>
  <body>
    <div id="content">
      %def content():
      %enddef
      {{content()}}
    </div>
    <div id="footer">
      %def footer():
      &copy; Copyright 2014 by Him.
      %enddef
      {{footer()}}
```

```
    </div>
  </body>
</html>
```

In this example, the `%def` tags define python functions (substitution areas). These functions are inserted into specific places (right after definition). These places become placeholders for child templates. The `%footer` placeholder defines default content while `%title` and `%content` are just empty.

Child Template

Child templates are used to extend master templates via placeholders defined:

```
%extends("shared/master.html")

%def title():
  Welcome
%enddef

%def content():
  <h1>Home</h1>
  <p>
    Welcome to My Site!
  </p>
%enddef
```

In this example, the `%title` and `%content` placeholders are overriden by the child template.

3.3.2 Include

The include is useful to insert a template content just in place of call:

```
%include("shared/snippet/script.html")
```

3.3.3 Import

The import is used to reuse some code stored in other files. So you are able to import all functions defined by that template:

```
%import "shared/forms.html" as forms

{{ forms.textbox('username') }}
```

or just a certain name:

```
%from "shared/forms.html" import textbox

{{ textbox(name='username') }}
```

Once imported you use these names as variables in the template.

3.4 Python code

Any line starting with a single `%` and which is not recognized as a directive is actual Python code. Its content is copied to the generated source code.

3.4.1 Line Statements

The `%import` and `%from` lines can be either directives or Python commands, depending on their arguments.

In addition to the special `%def`, all kinds of Python blocks are supported. The indentation is not significant, blocks must be ended explicitly.

- `%for`, `%if`, `%elif`, `%else`, `%while`: loops and conditionals
- `%try`, `%except`, `%else`, `%finally`: exception handlers
- `%end` identifies the end of the inner block. It is recommended to use the specific `%endfor`, `%endif`, `%endwhile` or `%endtry` directive, even if this rule is not strictly enforced
- for completeness `%class`/`%endclass` and `%with`/`%endwith` are also supported

The empty `%return` directive triggers an early return in the template. The code execution is stopped and the generated content is returned.

Here is a simple example:

```
%require(items)
%if items:
  %for i in items:
    {{i.name}}: {{i.price}}.
  %endfor
%else:
  No items found.
%endif
```

3.4.2 Line Comments

Only single line comments are supported.

The `%#` directive introduces a one-line comment. Comments are removed before the template is compiled.

```
%# TODO:
```

3.4.3 Line Join

In case you need to continue a long line without breaking it with new line during rendering use line join (`\`):

```
%if menu_name == active:
  <li class='active'> \
%else:
  <li> \
%endif
```

3.5 Restrictions

The line after the `%def` directive must not enter a new block (`%for`, `%if`, etc...). A workaround is to insert an empty comment line before opening the block.

The variables used in the template should be declared, either with a `%require` directive (recommended for templates loaded from the filesystem), or passed as keyword argument `(require=["nav", "body"])` when preparing the template (recommended for string templates). When using the `render_template()` function with a

`(source="...")` keyword argument, the declaration `%require` is automatically generated based on the names of the other keyword arguments passed to the function.

These features are not supported (among others):

- code blocks: as an alternative, prepend a `%` on each line
- multi-line comments: prepend `%#` on each line

CHAPTER 4

Fiole API

4.1 Decorators

fiole.**route**(*url*, *methods=('GET', 'HEAD')*, *callback=None*, *status=200*)
Register a method for processing requests.

fiole.**get**(*url*)
Register a method as capable of processing GET/HEAD requests.

fiole.**post**(*url*)
Register a method as capable of processing POST requests.

fiole.**put**(*url*)
Register a method as capable of processing PUT requests.

fiole.**delete**(*url*)
Register a method as capable of processing DELETE requests.

fiole.**errorhandler**(*code*)
Register a method for processing errors of a certain HTTP code.

4.2 Helpers

fiole.**send_file**(*request*, *filename*, *root=None*, *content_type=None*, *buffer_size=65536*)
Fetch a static file from the filesystem.

fiole.**get_template**(*name=None*, *source=None*, *require=None*)
Return a compiled template.

fiole.**render_template**(*template_name=None*, *source=None*, ***context*)
Render a template with values of the *context* dictionary.

fiole.**engine**
Default instance of the Template `Engine`.

fiole.**default_app**
Default `Fiole` application.

fiole.**get_app**()
Get the `Fiole` application which is on the top of the stack.

fiole.**run_wsgiref**(*host*, *port*, *handler*)
Simple HTTPServer that supports WSGI.

`fiole.`**`run_fiole`**(*app=default_app*, *server=run_wsgiref*, *host=None*, *port=None*)
Run the *Fiole* web server.

4.3 WSGI application

class `fiole.`**`Fiole`**
Web Application.

classmethod **`push`**(*app=None*)
Push a new `Fiole` application on the stack.

classmethod **`pop`**(*index=-1*)
Remove the `Fiole` application from the stack.

`debug`
Enable debugging: don't catch internal server errors (500) and unhandled exceptions. (default: *False*)

`secret_key`
Secret key used to sign secure cookies. (default: *unset*)

`static_folder`
Directory where static files are located. (default: *./static*)

`hooks`
List of *Hooks* which are registered for this application.

`handle_request`(*environ*, *start_response*)
The main handler. Dispatch to the user's code.

`handle_error`(*exception*, *environ*, *level=0*)
Deal with the exception and present an error page.

`find_matching_url`(*request*)
Search through the methods registered.

`route`(*url*, *methods=('GET', 'HEAD')*, *callback=None*, *status=200*)
Register a method for processing requests.

`get`(*url*)
Register a method as capable of processing GET/HEAD requests.

`post`(*url*)
Register a method as capable of processing POST requests.

`put`(*url*)
Register a method as capable of processing PUT requests.

`delete`(*url*)
Register a method as capable of processing DELETE requests.

`errorhandler`(*code*)
Register a method for processing errors of a certain HTTP code.

`encode_signed`(*name*, *value*)
Return a signed string with timestamp.

`decode_signed`(*name*, *value*, *max_age_days=31*)
Decode a signed string with timestamp or return `None`.

`send_file`(*request*, *filename*, *root=None*, *content_type=None*, *buffer_size=65536*)
Fetch a static file from the filesystem.

class `fiole.`**`Request`**(*environ*)
An object to wrap the environ bits in a friendlier way.

Environment variables are also accessible through `Request` attributes.

`environ`
Dictionary of environment variables

`path`
Path of the request, decoded and with `/` appended.

`method`
HTTP method (GET, POST, PUT, ...).

`query`
Read `QUERY_STRING` from the environment.

`script_name`
Read `SCRIPT_NAME` from the environment.

`host_url`
Build host URL.

`headers`
An instance of `EnvironHeaders` which wraps HTTP headers.

`content_length`
Header `"Content-Length"` of the request as integer or `0`.

`accept`
Header `"Accept"` of the request. Return an `Accept` instance.

`accept_charset`
Header `"Accept-Charset"` of the request. Return an `Accept` instance.

`accept_encoding`
Header `"Accept-Encoding"` of the request. Return an `Accept` instance.

`accept_language`
Header `"Accept-Language"` of the request. Return an `Accept` instance.

`GET`
A dictionary of GET parameters.

`POST`
A dictionary of POST (or PUT) values, including files.

`PUT`
A dictionary of POST (or PUT) values, including files.

`body`
Content of the request.

`cookies`
A dictionary of Cookie.Morsel objects.

`get_cookie`(*name*, *default=None*)
Get the value of the cookie with the given name, else default.

`get_secure_cookie`(*name*, *value=None*, *max_age_days=31*)
Return the given signed cookie if it validates, or None.

`get_url`(*path=''*, *full=False*)
Build the absolute URL for an application path.

By default it builds the current request URL with leading and trailing `/` and no query string. The boolean argument `full` builds a full URL, incl. host.

class `fiole.`**Response**(*output*, *headers=None*, *status=200*, *content_type='text/html'*, *wrapped=False*)

charset = 'utf-8'

status
Status code of the response as integer (default: *200*)

headers
Response headers as `HTTPHeaders`.

set_cookie(*name*, *value*, *domain=None*, *expires=None*, *path='/'*, *expires_days=None*, *signed=None*, ***kwargs*)
Set the given cookie name/value with the given options.

clear_cookie(*name*, *path='/'*, *domain=None*)
Delete the cookie with the given name.

set_secure_cookie(*name*, *value*, *expires_days=30*, ***kwargs*)
Sign and timestamp a cookie so it cannot be forged.

send(*environ*, *start_response*)
Send the headers and return the body of the response.

class `fiole.`**HTTPHeaders**(*headers=None*)
An object that stores some headers.

An instance of `HTTPHeaders` is an iterable. It yields tuples `(header_name, value)`. Additionally it provides a dict-like interface to access or change individual headers.

__getitem__(*name*)
Access the header by name. This method is case-insensitive and the first matching header is returned. It returns `None` if the header does not exist.

get(*name*, *default=None*)
Return the default value if the header doesn't exist.

get_all(*name*)
Return a list of all the values for the header.

add(*name*, *value*, ***kw*)
Add a new header tuple to the list.

set(*name*, *value*, ***kw*)
Remove all header tuples for *key* and add a new one.

setdefault(*name*, *value*)
Add a new header if not present. Return the value.

to_list(*charset='iso-8859-1'*)
Convert the headers into a list.

keys()

values()

items()

class `fiole.`**EnvironHeaders**(*environ*)
Headers from a WSGI environment. Read-only view.

__getitem__(*name*)
Access the header by name. This method is case-insensitive and returns `None` if the header does not exist.

get(*name*, *default=None*)
Return the default value if the header doesn't exist.

get_all(*name*)
Return a list of all the values for the header.

keys()

values()

items()

class `fiole.`**Accept**(*header_name*, *value*)
Represent an `Accept`-style header.

__contains__(*offer*)
Return True if the given offer is listed in the accepted types.

quality(*offer*)
Return the quality of the given offer.

best_match(*offers*, *default_match=None*)
Return the best match in the sequence of offered types.

exception `fiole.`**HTTPError**(*message*, *hide_traceback=False*)
Base exception for HTTP errors.

exception `fiole.`**BadRequest**(*message*, *hide_traceback=True*)

exception `fiole.`**Forbidden**(*message*, *hide_traceback=True*)

exception `fiole.`**NotFound**(*message*, *hide_traceback=True*)

exception `fiole.`**MethodNotAllowed**(*message*, *hide_traceback=True*)

exception `fiole.`**Redirect**(*url*)
Redirect the user to a different URL.

exception `fiole.`**InternalServerError**(*message*, *hide_traceback=False*)

4.4 Template engine

class `fiole.`**Engine**(*loader=None*, *parser=None*, *template_class=None*)
Assemble the template engine.

global_vars
This mapping contains additional globals which are injected in the generated source. Two special globals are used internally and must not be modified: `_r` and `_i`. The functions `str` and `escape` (alias `e`) are also added here. They are used as filters. They can be replaced by C extensions for performance (see Webext). Any object can be added to this registry for usage in the templates, either as function or filter.

default_filters
The list of filters which are applied to all template expressions `{{ ... }}`. Set to `None` to remove all default filters, for performance. (default: *['str']*)

clear()
Remove all compiled templates from the internal cache.

get_template(*name=None*, ***kwargs*)
Return a compiled template.

The optional keyword argument `default_filters` overrides the setting which is configured on the `Engine`.

remove(*name*)
Remove given `name` from the internal cache.

import_name(*name*, ***kwargs*)
Compile and return a template as module.

class `fiole.`**Template**
Simple template class.

name
Name of the template (it can be `None`).

render(*context*)
render(***context*)
Render the template with these arguments (either a dictionary or keyword arguments).

class `fiole.`**Loader**(*templates=None*)
Load templates.

`templates` - a dict where key corresponds to template name and value to template content.

template_folder
Directory where template files are located. (default: *./templates*)

list_names()
List all keys from internal dict.

load(*name*, *source=None*)
Return template by name.

class `fiole.`**Lexer**(*lexer_rules*)
Tokenize input source per rules supplied.

tokenize(*source*)
Translate `source` into an iterable of tokens.

class `fiole.`**Parser**(*token_start='%'*, *var_start='{{'*, *var_end='}}'*, *line_join='\'*)
Include basic statements, variables processing and markup.

tokenize(*source*)
Translate `source` into an iterable of tokens.

end_continue(*tokens*)
If token is `continue` prepend it with `end` token so it simulates a closed block.

parse_iter(*tokens*)
Process and yield groups of tokens.

class `fiole.`**BlockBuilder**(*indent=''*, *lineno=0*, *nodes=()*, *default_filters=None*)

filters = {'e': 'escape'}
A mapping of declared template filters (aliases of globals). This mapping can be extended. The globals can be extended too, see `Engine.global_vars`.

rules
A mapping of `tokens` with list of methods, to generate the source code.

add(*lineno*, *code*)
Add Python code to the source.

compile_code(*name*)
Compile the generated source code.

CHAPTER 5

Developer's notes

5.1 Frequently asked questions

5.1.1 Another web framework, are you kidding me?

Fiole is a new challenger in the category of the Python micro-frameworks. As you probably know, there are very good competitors in the market and we don't really need a new one.

Read *the introduction* for a quick outline of the development guidelines for the `Fiole` framework.

So, how `Fiole` is different or similar to the others?

- web.py is the first in the series, created in January 2006 by Aaron Swartz (aaronsw). It has no dependency and it supports old versions of Python, but it is not compatible with Python 3. It is a whole package and it provides additional features, while `Fiole` focuses on the essentials. `web.py` does not support the decorator syntax, compared with more recent frameworks.
- itty.py is a single-file micro-framework experiment, released on March 2009. It is the first Sinatra-influenced Python framework. However, it does not have public tests, and it does not ship a template engine.
- Bottle is a micro-web framework born on July 2009. `Fiole` is similar to `Bottle` because it is a single file, inspired by `itty.py`, with no dependency. It embeds a template engine, which has a syntax close to Bottle's SimpleTemplate. When it comes to the differences, `Fiole` template engine is faster, and its source code is smaller and follows the PEP 8 guidelines. On the other side `Bottle` has more features.
- Flask is the successor of Denied, born on 1st April 2010. It is a great framework with a must-read documentation. `Flask` is a package which depends on Werkzeug and Jinja. In contrast with it, `Fiole` is a single file without external dependencies and with a lot less documentation and less features. `Flask` has many extensions.

To sum up:

- `Fiole` is a single file like `itty.py` and `Bottle`
- `Fiole` has no dependency, same as `web.py`, `itty.py` and `Bottle`
- `Fiole` embeds a template engine similar to `web.py` and `Bottle`

- `Fiole` supports the decorator syntax like `itty.py`, `Bottle` and `Flask`
- `Fiole` supports signed cookies like `Flask` does
- `Fiole` source code is PEP8-compliant like `itty.py` and `Flask`
- `Fiole` supports Python 3 like `Bottle` and `Flask`
- `Fiole` supports hooks like `Bottle` and `Flask`
- `Fiole` *does not* have an extensive documentation like `Flask` or `Bottle`
- `Fiole` *does not* provide built-in adapters for every WSGI server like `itty.py` or `Bottle`

Of course the above comparison is partial and subjective.

5.1.2 How much is it extensible?

`Fiole` is thread-safe and you can configure more than one application (for complex projects).

`Fiole` supports hooks which can be registered for each application. Moreover, the components of `Fiole` are well thought to allow extensibility. For example the template engine is configurable through attributes, and all the components of the template engine can be subclassed easily.

As an alternative, you can use any template engine, such as `Jinja` or `Mako` instead of the built-in template engine. There's no specific integration between `Fiole` and the built-in template `Engine`.

Only the adapter for the `wsgiref` server is provided. You can write your own adapter for your preferred WSGI server. There are examples available in `Bottle` or `itty.py` source code for example.

5.1.3 How to make my application really fast?

First, I am not an expert about this topic. Still, there are various places where you can improve the performance of your web application. Some ideas that come to my mind:

- use a reverse proxy (`nginx`, ...)
- delegate serving static files to the proxy
- enable on-the-fly gzip compression on the proxy
- provide the relevant HTTP headers to enable browser caching
- replace the WSGI server with a scalable WSGI server which supports concurrency
- add server caching
- use load-balancing
- switch to PyPy

The template engine is already very fast. Even so, you can achieve a better performance with small changes:

- disable `default_filters` and use the `|str` filter only when needed
- replace the `escape` filter with a C implementation (e.g. `Webext`)

However the first thing to do is to benchmark your own application with realistic data in order to know where is the bottleneck before doing any random optimization.

5.1.4 I need only the template engine, can you release it?

Indeed, the *template engine* has some benefits: it is compact (~450 lines of code) and it is rather intuitive (basically, it's Python syntax). It is derived from the wheezy.template package which is very fast.

The template engine can be used to process any kind of text.

The good news is that the template engine is not bound to the web framework. Currently there's no plan to release it separately because `Fiole` is already a very small module and there's nothing wrong using only one of its two components: the *web framework* or the *template engine*.

5.2 Source code

The source code is available on GitHub under the terms and conditions of the *BSD license*. Fork away!

The tests are run against Python 2.7, 3.2 to 3.4 and PyPy on the Travis-CI platform.

Project on PyPI: https://pypi.python.org/pypi/fiole

5.3 Changes

5.3.1 0.4.1 (2014-07-03)

- Replace deprecated `cgi.escape` with a copy of Python 3.4's `html.escape`.

5.3.2 0.4 (2014-07-02)

- Add `Request.host_url`, `Request.script_name` and `Request.get_url(path, full=False)`. (Issue #4)
- Add `|n` filter to disable default filters. (Issue #5)
- Fix caching of `Request.accept*` headers.

5.3.3 0.3 (2013-06-12)

- Improve the documentation.
- Add the Fiole application to the WSGI environment: `environ['fiole.app']`. (Issue #1)
- Implement parsing of the `Accept` headers, and add them as dynamic properties of `Request`: `accept`, `accept_charset`, `accept_encoding` and `accept_language`. (Issue #2)
- Replace the global `SECRET_KEY` with a new attribute of the Fiole application `app.secret_key`.
- Replace the helpers `_create_signed_value` and `_decode_signed_value` with methods: `Fiole.encode_signed()` and `Fiole.decode_signed()`. The method `Response.create_signed_value()` is removed too.
- Remove argument `secret` from the `run_fiole` function: use `get_app().secret_key = 's3c4e7k3y...'` instead.
- The `send_file` helper recognizes the `If-Modified-Since` header and returns *"304 Not Modified"* appropriately.

- Patch the `wsgiref.simple_server.ServerHandler` to stop sending `Content-Length` for status *"304 Not Modified"*. (This is related to a Python bug)
- Add `Fiole.debug` boolean flag to let unhandled exceptions propagate.
- Rename helper `html_escape` to `escape_html`.
- Add `default_filters` for the template engine configuration.
- Automatically cast Python objects to Unicode for template rendering. This can be disabled with `engine.default_filters = None`.
- Refactor the internals of the template engine. New method `Engine.clear()` to reset the cache of byte-compiled templates.
- Support extensibility of the WSGI application with hooks.

5.3.4 0.2 (2013-05-22)

- Initial release.

CHAPTER 6

Indices and tables

- *genindex*
- *search*

CHAPTER 7

Credits

Project created by Florent Xicluna.

Thank you to Daniel Lindsley (toastdriven) for itty, the itty-bitty web framework which helped me to kick-start the project.

Thank you to Andriy Kornatskyy (akorn) for his blazingly fast and elegant template library wheezy.template: it is the inspiration for the template engine of `fiole.py`.

The following projects were also a great source of ideas:

- Werkzeug (`HTTPHeaders` and `EnvironHeaders` datastructures)
- WebOb (parsing the `Accept` headers)
- Bottle (embedding a simple template engine)
- Jinja2 and Mako (common template engine syntax and features)

CHAPTER 8

License

This software is provided under the terms and conditions of the BSD license:

```
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
#
#   * Redistributions of source code must retain the above copyright
#     notice, this list of conditions and the following disclaimer.
#
#   * Redistributions in binary form must reproduce the above copyright
#     notice, this list of conditions and the following disclaimer in the
#     documentation and/or other materials provided with the distribution.
#
#   * The names of the contributors may not be used to endorse or
#     promote products derived from this software without specific
#     prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT OWNERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNERS OR CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
# THE POSSIBILITY OF SUCH DAMAGE.
```

f