aiohttp_tal Documentation

Release 0.1

Aleix Llusà Serra

Apr 05, 2019

Contents

TAL Chameleon template engine renderer for aiohttp.web. Based on aiohttp_jinja2.

CHAPTER 1

Installation

Install from PyPI:

```
pip install aiohttp-tal
```

CHAPTER 2

Developing

Install requirement and launch tests:

```
pip install -r requirements-dev.txt
pytest tests
```

CHAPTER 3

Usage

For more details on usage, see https://aiohttp-tal.readthedocs.io/en/latest/usage.html.

Before template rendering you have to setup *TAL environment* first:

```
app = web.Application()
aiohttp_tal.setup(app,
    loader=chameleon.PageTemplateLoader('/path/to/templates/folder'))
```

Import:

```
import aiohttp_tal
import chameleon
```

After that you may to use template engine in your *web-handlers*. The most convenient way is to decorate a *web-handler*.

Using the function based web handlers:

```
@aiohttp_tal.template('tmpl.pt')
def handler(request):
    return {'name': 'Andrew', 'surname': 'Svetlov'}
```

CHAPTER 4

License

`aiohttp_tal` is offered under the GPLv3 license.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

CHAPTER 5

Source code

The project is hosted on GitHub.

Please feel free to file an issue on bug tracker if you have found a bug or have some suggestion for library improvement.

The project uses Travis for Continuous Integration.

CHAPTER 6

Contents

6.1 Usage

aiohttp_tal has the same API as aiohttp_jinja2. See https://aiohttp-jinja2.readthedocs.io/en/stable/#reference for more information.

It uses chameleon as template loader. See *chameleon.PageTemplate** template classes, e.g:

- chameleon.PageTemplate for TAL string input (default for *string* input)
- chameleon.PageTemplateFile for TAL filename input
- chameleon.chameleon.PageTemplateLoader for directory of TAL templates input

6.1.1 Initialization

Simple initialization:

```
import chameleon
import aiohttp_tal
from aiohttp import web
from pathlib import Path


THIS_DIR = Path(__file__).parent

app = web.Application()
loader = chameleon.PageTemplateLoader(str(THIS_DIR / 'templates'))
aiohttp_tal.setup(app, loader=loader)
```

where *loader* could also be a *dict* of template name and TAL input:

```
loader = {'tmpl.pt': chameleon.PageTemplate('<html>${text}</html>')}
```

or directly the TAL *string* input:

```
loader =  {'tmpl.pt': '<html>${text}</html>'}
```

6.1.2 Rendering

Based on https://aiohttp-jinja2.readthedocs.io/en/stable/#usage and https://aiohttp-jinja2.readthedocs.io/en/stable/#default-globals.

After *initializing *TAL environment**, you may use template engine in your *web-handlers*. The most convenient way is to decorate a *web-handler*.

Using the function based web handlers:

```
@aiohttp_tal.template('tmpl.pt')
def handler(request):
    return {'name': 'Andrew', 'surname': 'Svetlov'}
```

Or the class-based views (`aiohttp.web.View`):

```
class Handler(web.View):
    @aiohttp_tal.template('tmpl.pt')
    async def get(self):
        return {'name': 'Andrew', 'surname': 'Svetlov'}
```

On handler call the `template()` decorator will pass returned dictionary `{'name': 'Andrew', 'surname': 'Svetlov'}` into template named `"tmpl.pt"` for getting resulting HTML text.

If you need more complex processing (set response headers for example) you may call `render_template()` function.

Using a function based web handler:

```
async def handler(request):
    context = {'name': 'Andrew', 'surname': 'Svetlov'}
    response = aiohttp_tal.render_template('tmpl.pt',
                                           request,
                                           context)
    response.headers['Content-Language'] = 'ru'
    return response
```

Or, again, a class-based view (`aiohttp.web.View`):

```
class Handler(web.View):
    async def get(self):
        context = {'name': 'Andrew', 'surname': 'Svetlov'}
        response = aiohttp_tal.render_template('tmpl.pt',
                                               self.request,
                                               context)
        response.headers['Content-Language'] = 'ru'
        return response
```

Context processors is a way to add some variables to each template context. It works like `aiohttp_tal.Environment().globals`, but calculate variables each request. So if you need to add global constants it will be better to use `aiohttp_tal.Environment().globals` directly. But if you variables depends of request (e.g. current user) you have to use context processors.

Context processors is following last-win strategy. Therefore a context processor could rewrite variables delivered with previous one.

In order to use context processors create required processors:

```
async def foo_processor(request):
    return {'foo': 'bar'}
```

And pass them into `setup()`:

```
aiohttp_tal.setup(
    app,
    context_processors=[foo_processor,
                        aiohttp_tal.request_processor],
    loader=loader)
```

As you can see, there is a built-in `request_processor()`, which adds current `aiohttp.web.Request` into context of templates under `'request'` name.

Here is an example of how to add current user dependant logic to template (requires `aiohttp_security` library):

```
from aiohttp_security import authorized_userid

async def current_user_ctx_processor(request):
    userid = await authorized_userid(request)
    is_anonymous = not bool(userid)
    return {'current_user': {'is_anonymous': is_anonymous}}
```

Template:

```
<body>
    <div>
        <a tal:condition="current_user.is_anonymous" href="${url('login')}">Login</a>
        <a tal:condition="not:current_user.is_anonymous"  href="${url('logout')}">
↪Logout</a>
    </div>
</body>
```

Default Globals

`app` is always made in templates via `aiohttp_tal.Environment().globals`:

```
<body>
    <h1>Welcome to ${app['name']}</h1>
</body>
```

Two more helpers are also enabled by default: `url` and `static`.

`url` can be used with just a view name:

```
<body>
    <a href="${url('index')}">Index Page</a>
</body>
```

Or with arguments:

```
<body>
    <a href="${url('user', id=123)}">User Page</a>
</body>
```

A query can be added to the url with the special `query_` keyword argument:

```
<body>
    <a href="${url('user', id=123, query_={'foo': 'bar'})}">User Page</a>
</body>
```

For a view defined by `app.router.add_get('/user-profile/{id}/', user, name='user')`, the above would give:

```
<body>
    <a href="/user-profile/123/?foo=bar">User Page</a>
</body>
```

This is useful as it would allow your static path to switch in deployment or testing with just one line.

The `static` function has similar usage, except it requires you to set `static_root_url` on the app

```
app = web.Application()
aiohttp_tal.setup(app,
    loader=chameleon.PageTemplateLoader('/path/to/templates/folder'))
app['static_root_url'] = '/static'
```

Then in the template:

```
<script src="${static('dist/main.js')}"></script>
```

Would result in:

```
<script src="/static/dist/main.js"></script>
```

Both `url` and `static` can be disabled by passing `default_helpers=False` to `aiohttp_tal.setup`.

6.2 Examples

Download full examples code from GitHub.

6.2.1 Simple

A simple TAL example with METAL macros.

Install:

```
pip install aiohttp-tal
```

Run:

```
python simple.py
```

Listing 1: simple.py

```
from pathlib import Path

from aiohttp import web
import aiohttp_tal
from chameleon import PageTemplateLoader

```

(continues on next page)

(continued from previous page)

```
THIS_DIR = Path(__file__).parent


@aiohttp_tal.template('index.html')
async def index(request):
    # Note: we return a dict not a response because of the @template decorator
    return {
        'title': request.app['name'],
        'intro': "Success! you've setup a basic aiohttp app with TAL.",
        }


@aiohttp_tal.template('translation.html')
async def page(request):
    return {
        'title': 'First page',
        }


async def create_app():
    app = web.Application()
    app.update(name='Testing aiohttp TAL')

    tal_loader = PageTemplateLoader(str(THIS_DIR / 'templates'),
                                    auto_reload=True  # debugging
                                    )
    aiohttp_tal.setup(app, loader=tal_loader)

    app.add_routes([web.static('/static', str(THIS_DIR / 'static'))])
    app['static_root_url'] = '/static'
    app.router.add_get('/', index, name='index')
    app.router.add_get('/page', page, name='translation')

    return app


if __name__ == '__main__':
    web.run_app(create_app(), host='127.0.0.1', port=8080)
```

Macro

Listing 2: templates/base.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-us"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      metal:define-macro="master">
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>${app.name}</title>
  <link href="../static/style.css" tal:attributes="href static('style.css')" rel=
↪"stylesheet" />
</head>
```

(continues on next page)

(continued from previous page)

```
<body>
  <main>
    <h1>${title}</h1>
    <div metal:define-slot="content"></div>
  </main>
</body>
</html>
```

Page template

Listing 3: templates/index.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      metal:use-macro="load: base.html">
<head>
  <link href="../static/style.css" rel="stylesheet" tal:condition="nothing" />
</head>
<body>
<div metal:fill-slot="content">
  <p>${intro}</p>

  <p>
    To demonstrate a little of the functionality of aiohttp TAL.
  </p>
  <b>
    <a href="translation.html" tal:attributes="href url('translation')">View first␣
↪page</a>
  </b>
</div>
</body>
</html>
```

6.2.2 Translation (I18N)

See TAL I18N and aiohttp_babel.

Install:

```
pip install aiohttp-tal aiohttp_babel babel-lingua-chameleon
```

Run:

```
python translation.py
```

Listing 4: translation.py

```
"""

First time:

* pybabel extract -F babel.cfg -o locales/mydomain.pot .
* pybabel init -D mydomain -i locales/mydomain.pot -d locales -l ca
```

(continues on next page)

(continued from previous page)

```
* pybabel init -D mydomain -i locales/mydomain.pot -d locales -l fr_fr
* pybabel compile -D mydomain -d locales

Updates:

* pybabel extract -F babel.cfg -o locales/mydomain.pot .
* pybabel update -D mydomain -i locales/mydomain.pot -d locales
* pybabel compile -D mydomain -d locales

"""
from pathlib import Path

from aiohttp import web
from aiohttp_babel.locale import load_gettext_translations, set_default_locale
from aiohttp_babel.middlewares import babel_middleware, _
import aiohttp_tal
from chameleon import PageTemplateLoader


THIS_DIR = Path(__file__).parent

set_default_locale('en_GB')
load_gettext_translations(str(THIS_DIR / 'locales'), 'mydomain')


@aiohttp_tal.template('index.html')
async def index(request):
    return {
        'title': request.app['name'],
        'intro': "Success! you've setup a basic aiohttp app with TAL.",
        }


@aiohttp_tal.template('translation.html')
async def translation(request):
    return {
        'title': _('First page'),
        'request': request,
        }


def translate(msgid, domain=None, mapping=None, context=None,
              target_language=None, default=None):
    # _(message, plural_message=None, count=None, **kwargs):
    return str(_(msgid))


async def create_app():
    app = web.Application(middlewares=[babel_middleware])
    app.update(name='Testing aiohttp TAL')

    tal_loader = PageTemplateLoader(str(THIS_DIR / 'templates'),
                                    translate=translate,
                                    auto_reload=True  # debugging
                                    )
    aiohttp_tal.setup(app, loader=tal_loader)
```

(continues on next page)

(continued from previous page)

```
    app.add_routes([web.static('/static', str(THIS_DIR / 'static'))])
    app['static_root_url'] = '/static'
    app.router.add_get('/', index, name='index')
    app.router.add_get('/translation', translation, name='translation')

    return app


if __name__ == '__main__':
    web.run_app(create_app(), host='127.0.0.1', port=8080)
```

Page template with i18n

Listing 5: templates/translation.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      metal:use-macro="load: base.html"
      i18n:domain="mydomain">
<head>
  <link href="../static/style.css" tal:attributes="href static('style.css')" rel=
↪"stylesheet" tal:condition="nothing"/>
</head>
<body>
<div metal:fill-slot="content">


<p i18n:translate="">The first page</p>

<p tal:condition="request.locale|nothing">
  <span tal:define="datetime import: datetime;
                    today datetime.datetime.now();"
        tal:content="request.locale.format_datetime(today)">
        1/1/1970
  </span>
</p>


<b>
<a href="index.html" tal:attributes="href url('index')"
   title="Go to index" i18n:attributes="title"
   i18n:translate="">
  View Index
</a>
</b>


<hr/>

<p tal:condition="request.locale|nothing">
<a href="" onclick="document.cookie='locale=ca'">ca</a>
<a href="" onclick="document.cookie='locale=fr_FR'">fr</a>
```

(continues on next page)

(continued from previous page)

```
<a href="" onclick="document.cookie='locale=en'">en</a>
</p>

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

6.3 CHANGES

6.3.1 0.2.dev (unreleased)

6.3.2 0.1.0 (2019-03-28)

- Initial release. Based on aiohttp-jinja2 copyright by Andrew Svetlo and aio-libs team.

CHAPTER 7

Glossary

METAL Macro Expansion TAL

TAL Template Attribute Language. See https://chameleon.readthedocs.io/en/latest/reference.html

TALES TAL Expression Syntax

web-handler An endpoint that returns http response.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index