aiohttp_tal Documentation

Release 0.1

Aleix Llusà Serra

Mar 29, 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 CHANGES

6.2.1 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