# aiohttp_jinja2 Documentation

*Release 1.5.None-None*

**Andrew Svetlov**

**Feb 01, 2023**

# CONTENTS

*jinja2* template renderer for aiohttp.web.

CHAPTER

# ONE

# USAGE

Before template rendering you have to setup *jinja2 environment* (`jinja2.Environment`) first:

```
app = web.Application()
aiohttp_jinja2.setup(app,
    loader=jinja2.FileSystemLoader('/path/to/templates/folder'))
```

After that 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_jinja2.template('tmpl.jinja2')
def handler(request):
    return {'name': 'Andrew', 'surname': 'Svetlov'}
```

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

```
class Handler(web.View):
    @aiohttp_jinja2.template('tmpl.jinja2')
    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.jinja2"` for getting resulting HTML text.

More complex template processing can be achieved by modifying the existing list of global functions. Modification of Jinja2's environment can be done via *get_env()*. For example, adding the `zip` function:

```
env = aiohttp_jinja2.get_env(app)
env.globals.update(zip=zip)
```

Which can now to be used in any template:

```
{% for value, square in zip(values, squares) %}
    <p>The square of {{ value }} is {{ square }}.</p>
{% endfor %}
```

In some cases, finer control over the dataflow may also be required. This can be worked out by explicitly asking for template to be rendered using *render_template()*. Explicit rendering will allow to possibly pass some context to the renderer and also to modify its response on the fly. This can for example be used to set response headers:

```
async def handler(request):
    context = {'name': 'Andrew', 'surname': 'Svetlov'}
```

(continues on next page)

(continued from previous page)

```
    response = aiohttp_jinja2.render_template('tmpl.jinja2',
                                              request,
                                              context)
    response.headers['Content-Language'] = 'ru'
    return response
```

This, again, can also be done with a class-based view (`aiohttp.web.View`):

```
class Handler(web.View):
    async def get(self):
        context = {'name': 'Andrew', 'surname': 'Svetlov'}
        response = aiohttp_jinja2.render_template('tmpl.jinja2',
                                                  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 `jinja2.Environment().globals`, but calculate variables each request. So if you need to add global constants it will be better to use `jinja2.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_jinja2.setup(
    app,
    context_processors=[foo_processor,
                        aiohttp_jinja2.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>
```

(continues on next page)

(continued from previous page)

```
        {% if current_user.is_anonymous %}
            <a href="{{ url('login') }}">Login</a>
        {% else %}
            <a href="{{ url('logout') }}">Logout</a>
        {% endif %}
    </div>
</body>
```

## 1.1 Async functions

If you pass the `enable_async` parameter to the setup function, then you will need to use the async functions for rendering:

```
aiohttp_jinja2.setup(
    app, enable_async=True,
    loader=jinja2.FileSystemLoader('/path/to/templates/folder'))

...

async def handler(request):
    return await aiohttp_jinja2.render_template_async(
        'tmpl.jinja2', request)
```

The `@aiohttp_jinja2.template` decorator will work for both cases.

## 1.2 Default Globals

`app` is always made in templates via `jinja2.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_jinja2.setup(app,
    loader=jinja2.FileSystemLoader('/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_jinja2.setup`.

CHAPTER

TWO

# LIBRARY INSTALLATION

The *aiohttp_jinja2* can be installed by pip:

```
$ pip3 install aiohttp_jinja2
```

CHAPTER

THREE

# 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 library uses Travis for Continuous Integration.

CHAPTER

# FOUR

# LICENSE

*aiohttp_jinja2* is offered under the Apache 2 license.

CHAPTER

FIVE

# CONTENTS

## 5.1 API

Describes the module API with detailed explanations of functions parameters.

### 5.1.1 APP_KEY

aiohttp_jinja2.**APP_KEY**

The key name in `aiohttp.web.Application` dictionary, `'aiohttp_jinja2_environment'` for storing *jinja2* environment object (`jinja2.Environment`).

Usually you don't need to operate with *application* manually, left it to *aiohttp_jinja2* functions.

### 5.1.2 setup

aiohttp_jinja2.**setup**(*app*, **args*, *app_key=APP_KEY*, *context_processors=()*, *autoescape=True*, *filters=None*, *default_helpers=True*, ***kwargs*)

Function responsible for initializing templating system on application. It must be called before freezing or running the application in order to use *aiohttp-jinja*.

**Parameters**

- **app** – `aiohttp.web.Application` instance to initialize template system on.
- **app_key** (*str*) – optional key that will be used to access templating environment from application dictionary object. Defaults to *aiohttp_jinja2_environment*.
- **context_processors** (`list`) – list of Middlewares. These are context processors used to rewrite or inject some variables during the processing of a request.
- **autoescape** – the argument is passed to `jinja2.Environemnt`, see *Autoescaping* for more details.
- **filters** (`list`) – extra jinja filters (*link to docs <http://jinja.pocoo.org/docs/2.10/templates/#filters>*).
- **default_helpers** (*bool*) – whether to use default global helper in templates provided by package `aiohttp_jinja2.helpers` or not.
- ***args** – positional arguments passed into environment constructor.
- ****kwargs** – any arbitrary keyword arguments you want to pass to `jinja2.Environment` environment.

Simple initialization:

```
import jinja2
import aiohttp_jinja2
from aiohttp import web

app = web.Application()
aiohttp_jinja2.setup(
   app,
   loader=jinja2.FileSystemLoader('/path/to/templates/folder'),
)
```

### 5.1.3 @template

@aiohttp_jinja2.**template**(*template_name*, *, *app_key=APP_KEY*, *encoding='utf-8'*, *status=200*)

Behaves as a decorator around view functions accepting template name that should be used to render the response. Supports both synchronous and asynchronous functions.

**Parameters**

- **template_name** (*str*) – name of the template file that will be looked up by the loader. Raises a 500 error in case template was not found.
- **app_key** (*str*) – optional key that will be used to access templating environment from application dictionary object. Defaults to *aiohttp_jinja2_environment*.
- **encoding** (*str*) – encoding that will be set as a charset property on the response for rendered template, default to utf-8.

**Params int status** http status code that will be set on resulting response.

Simple usage example:

```
@jinja2.template('tmpl.jinja2')
async def handler(request):
   context = {'foo': 'bar'}
   return context

app.router.add_get('/tmpl', handler)
```

### 5.1.4 render_string

aiohttp_jinja2.**render_string**(*template_name*, *request*, *context*, *, *app_key=APP_KEY*)

Renders template specified and returns resulting string.

**Parameters**

- **template_name** (*str*) – Name of the template you want to render. Usually it's a filename without extension on your filesystem.
- **request** (aiohttp.web.Request) – aiohttp request associated with an application where aiohttp-jinja rendering is configured.
- **context** (*dict*) – dictionary used as context when rendering the template.
- **app_key** (*str*) – optional key that will be used to access templating environment from application dictionary object. Defaults to *aiohttp_jinja2_environment*.

### 5.1.5 render_string_async

async aiohttp_jinja2.**render_string_async**(*template_name*, *request*, *context*, *, *app_key=APP_KEY*)

Async version of `render_string()`.

Replaces `render_string()` when `enable_async=True` is passed to the `setup()` call.

See `render_string()` for parameter usage.

### 5.1.6 render_template

aiohttp_jinja2.**render_template**(*template_name*, *request*, *context*, *, *app_key=APP_KEY*, *encoding='utf-8'*, *status=200*)

**Parameters**

- **template_name** (*str*) – Name of the template you want to render.
- **request** (aiohttp.web.Request) – aiohttp request associated with an application where aiohttp-jinja rendering is configured.
- **context** (*dict*) – dictionary used as context when rendering the template.
- **app_key** (*str*) – optional key that will be used to access templating environment from application dictionary object. Defaults to *aiohttp_jinja2_environment*.
- **status** (*int*) – http status code that will be set on resulting response.

Assuming the initialization from the example above has been done:

```
async def handler(request):
   context = {'foo': 'bar'}
   response = aiohttp_jinja2.render_template('tmpl.jinja2',
                                             request,
                                             context)
   return response

app.router.add_get('/tmpl', handler)
```

### 5.1.7 render_template_async

async aiohttp_jinja2.**render_template_async**(*template_name*, *request*, *context*, *, *app_key=APP_KEY*, *encoding='utf-8'*, *status=200*)

Async version of `render_template()`.

Replaces `render_template()` when `enable_async=True` is passed to the `setup()` call.

See `render_template()` for parameter usage.

aiohttp_jinja2.**get_env**(*app*, *, *app_key=APP_KEY*)

Get aiohttp-jinja2 environment from an application instance by key.

**Parameters**

- **app** – aiohttp.web.Application instance to get variables from.
- **app_key** (*str*) – optional key that will be used to access templating environment from application dictionary object. Defaults to *aiohttp_jinja2_environment*.

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- search

CHAPTER

SEVEN

# GLOSSARY

**jinja2** A modern and designer-friendly templating language for Python.

See http://jinja.pocoo.org/

**web-handler** An endpoint that returns http response.

# PYTHON MODULE INDEX

# INDEX