Axe Documentation

Release 0.0.1

soasme

May 06, 2014

Contents

Welcome to Axe's documentation.

CHAPTER 1

User's Guide

1.1 Quickstart

This page gives a good introduction to Axe. You should have Axe installed first.

1.1.1 A Minimal Application

A minimal Axe application looks something like this:

```
from axe import Axe
app = Axe()

def index():
    return 'Hello World!'

urls = {
    '/': index,
}
app.build(urls)

if __name__ == '__main__':
    app.run_simple(host='127.0.0.1', port=8384)
```

You can save it as *hello_world.py*:

```
$ python hello_world.py
```

Now go to your browser and visit http://127.0.0.1:8384/, and you should see *Hello world!* in your screen.

To stop the server, hit Ctrl-C.

WARNING: *run_simple* is for local development. It's strongly recommend not to use in production environment.

1.1.2 Debug Mode

If you pass *use_debugger=True* parameter to run_simple, You will have an excellent debug stacktrace when the page occur error:

```
app.run_simple(use_debugger=True)
```

If you pass *use_reloader=True* parameter to run_simple, the server will auto restart whenever a file modified:

```
app.run_simple(use_reloader=True)
```

More information about run_simple, see Werkzeug Documentation of run_simple

1.1.3 Routing

1.1.4 Static Serve

1.1.5 Template

It's you choice to use which template engine: Mako, Plim, Haml, Jinja, etc. There is no default template engine now.

1.1.6 Dependency Injection

The route controller functions always have many dependencies: query, form, json, headers or any other specific of you project. But it's hard to debug if you attach too much values in one *request* object. Here is the solution of *Axe*: DI(Dependency Injection). List all the dependencies as parameter in controller function, and happy to use them. We call these dependencies as extension in *Axe*. There are several default extensions like *query*, *json*, *form*, *headers*, *request*, *method*. But *Axe* enable you to write your own extensions.

1.1.7 Redirects and Errors

Use *axe.redirect* to direct the page:

```
from axe import redirect, ext

@ext
def require_login(session):
    if not session:
        return redirect('/login')

def index(require_login):
    return template('index.html')

def login():
    return template('login')

app.build({
    '/': index,
    '/login': login,
})
```

Use *axe.error* to define the error action:

```
from axe import error

@error(404)
def not_found(exc):
    return template('not_found.html')
```

1.1.8 About Response

1.1.9 Sessions

1.1.10 Logging

1.1.11 Scale Application

When your project becomes big, it's better to split it into several small projects. *Axe* allow you to assemble several WSGI application together when needed:

```
from MyVanillaApiV1 import v1
from MyVanillaApiV2 import v2
from MyVanillaWeb import web
app = Axe()
app.proxy({
    '/api/1': v1,
    '/api/2': v2,
    '/': web,
})
```

1.2 Extensions

The purpose of Axe extensions is to provide a readable baseline which makes view function can reliably and transparently execute. Axe extension mechanism offer dramatic improvements over attaching values with global context.

- Extensions have explicit names and are activated only when need it.
- Each extension name triggers an extension function which can itself use other extensions.

1.2.1 Extensions as View Function arguments

View functions can receive extension objects by naming them as an input argument. For each argument name, an extension function with that name provides the extension object. After initialize Axe app, *app = Axe()*, Extension functions as registered by decorating them with *@app.ext*. Let's look at a simple self-contained Axe app containing a customized extension:

```
from axe import Axe
import os
app = Axe()

@app.ext
def config():
    return {'system': os.name}

def index(config):
    return config.get('system', 'Unknown')

app.build({'/': index})

if __name__ == '__main__':
    app.run_simple()
```

Here, the *index* view function needs the *config* extension value. Axe app will discover all the extensions that registered to it and call the *@app.ext* marked *config* extension function. Running this example, and visit '/':

```
~ % curl http://localhost:8384
posix
```

You might got a different result, but that's trivial. Here is the exact process executed by Axe to call view function this way:

1. curl make a request of '/', and Axe app route '/' to *index* view function.
2. *index* view function needs a function argument named *config*. A matching extension is discovered by looking for an extension-decorated function named *config*.
3. *config()* is called and return a dict result.
4. *index({'system': 'posix'})* is actully called and the rest is view function logic: get key *system* in *config* dict as response body.

Note that if you misspell a function argument or want to use one that isn't available, you will see an error *axe.errors.UnrecognizedExtension* before app running, alas, the app is failed to start.

1.2.2 Sharing an extension across view functions

The extension can be applied into all view functions that is built by *app.build*. Multiple view functions after building will each receive the same extension function, and build it within every request.

1.2.3 Chain

You can not only use extensions in view functions but extension functions can use other extensions themselves. Here is a default extension *json* offered by Axe:

```
@app.ext
def json(headers, body):
    content_type = headers.get('Content-Type')
    if content_type != 'application/json':
        return
    data = body.decode('utf8')
    try:
        return json.loads(data)
    except ValueError:
        raise BadJSON
```

Note that avoid writing circular dependency for extensions.

1.2.4 Modularity

You might got mad by writing many input parameters in a view function. As we have ability to chaining extensions, Here is a simple example for you to extend the previous *config* example. We instantiate an object *exts* where we stick the already defined *config* resource into it:

```
class Exts(object):
    def __init__(config):
        self.config = config

@app.ext
def exts(config):
```

```
    return Exts(config)

def index(exts):
    return exts.config.get('system')
```

1.2.5 Default Extensions

Query

query extension return a *dict* object that contains key-value map from querystring like */hello?name=world*. Default value is *{}*. Example:

```
def hello(query):
    return query.get('name', '')
```

Form

form extension return a *dict* object that coming from form submitted from form. Default value is *{}*, Example:

```
def comment(form):
    Comment.create(form['email'], form['name'], form['content'])
```

Body

body extension return a string which composed request body. Example:

```
def resp_body(body):
    return body

$ curl http://localhost:8384/resp_body -d "This is body."
This is body.
```

Cookies

cookies extension return a dict object that is parsed from header *HTTP_COOKIE*.

Headers

headers extension return a dict object that is parsed from request headers. Example:

```
@app.ext
def auth(headers):
    token = headers.get('Authorization', '')
    if not (token.startswith('Bearer ') and Token.verify(token)):
        raise InvalidAuthorationToken(token)
    return Token.get_user_from_token(token)
```

JSON

json extension return a *dict* object only if there is request header *Content-Type: application/json* with request body in legal JSON encoding. If body is not in valid JSON format, Axe will response 400 Bad Request. Default value is *None*.:

```
def share(json, auth):
    if 'facebook' in json:
        share_to_facebook(auth, json['content'])
    if 'twitter' in json:
        share_to_twitter(auth, json['content'])
    return 204
```

Method

method extension return a word in upper case, choices: (*GET*, *POST*, *DELETE*, *PUT*, *OPTIONS*, *HEAD*).

1.3 Grow

Here are your options when growing your codebase or scaling your application.

1.3.1 Extension

It's a good idea to write your awesome extensions. Axe is just a skeleton, it's your choice to inject which kind of soul.

1.3.2 Proxy

Split an enormous codebase into several small and delicate codebases. It's good to keep codebase in a proper size. You may run many progresses and use server applications, like Nginx, Apache, to dispatch request to the right process. Or you might want them to work in the same progress: just combine them by the proxy method into a larger one based on prefix. See *Scale Application*.