zunzuncito Documentation

Release 0.1.16

Nicolas Embriz

Sep 27, 2017

Contents

micro-framework for creating REST API's.

CHAPTER 1

Table of Contents

Zunzuncito

micro-framework for creating REST API's.

Design Goals

- Keep it simple and small, avoiding extra complexity at all cost. KISS
- Create routes on the fly or by defining regular expressions.
- Support API versions out of the box without altering routes.
- Thread safety.
- Via decorator or in a defined route, accept only certain HTTP methods.
- Follow the single responsibility principle.
- Be compatible with any WSGI server, example: uWSGI, Gunicorn, Twisted, etc.
- Tracing Request-ID "rid" per request.
- Compatibility with Google App Engine. demo
- Multi-tenant Support.
- Ability to create almost anything easy, example: Support chunked transfer encoding.

What & Why ZunZuncito

ZunZuncito is a python package that allows to create and maintain REST API's without hassle.

The simplicity for sketching and debugging helps to develop very fast; versioning is inherit by default, which allows to serve and maintain existing applications, while working in new releases without need to create separate instances. All the applications are WSGI PEP 333 compliant, allowing to migrate existing code to more robust frameworks, without need to modify the existing code.

Why ?

"The need to upload large files by chunks and support resumable uploads trying to accomplish something like the nginx upload module *does in pure python."*

The idea of creating ZunZuncito, was the need of a very small and light tool (batteries included), that could help to create and deploy REST API's quickly, without forcing the developers to learn or follow a complex flow but, in contrast, from the very beginning, guide them to properly structure their API, giving special attention to "versioned URI's", having with this a solid base that allows to work in different versions within a single ZunZun instance without interrupting service of any existing API resources.

Download

Zip File: https://github.com/nbari/zunzuncito/zipball/master

Tar Ball: https://github.com/nbari/zunzuncito/tarball/master

View on GitHub: https://github.com/nbari/zunzuncito

See also:

Install

Install

The easy way is by using pip:

```
$ pip install zunzuncito
```

If you don't have pip, after downloading the sources, you can run:

```
$ python setup.py install
```

Clone the repository:

```
$ git clone https://github.com/nbari/zunzuncito.git
```

Quick Start

This is the directory structure:

API directory structure

app.py application python file.

my_api The **root** directory of the API.

default The **vroot** directory.

v0 Directory for default API resources or for version **0** when specified.

v1 Directory for API resources version **1**

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
        |--__init__.py
        `--default
           |--__init__.py
           |--v0
           |   |--__init__.py
           |   |--zun_default
           |   |   |--__init__.py
           |   |   `--zun_default.py
           |   `--zun_hasher
           |       |--__init__.py
           |       `--zun_hasher.py
           `--v1
               |--__init__.py
               |--zun_default
               |   |--__init__.py
               |   `--zun_default.py
               `--zun_hasher
                   |--__init__.py
                   `--zun_hasher.py
```

Inside directory /home/zunzun there is a file called **app.py** and a directory **my_api**.

For a very basic API, contents of file **app.py** can be:

```
1   import zunzuncito
2
3   root = 'my_api'
4
5   versions = ['v0', 'v1']
6
7   hosts = {'*': 'default'}
8
9   routes = {'default':[
10      ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST'),
11      ('/.*', 'default')
12  ]}
13
14  app = zunzuncito.ZunZun(root, versions, hosts, routes)
15
16  # For appending the Request-ID header on GAE
17  # app = zunzuncito.ZunZun(root, versions, hosts, routes, rid='REQUEST_LOG_ID')
```

- line 3 defines the "document root" for your API
- line 7 gives multitenant support, in the example all "*" is going to be handled by the 'default' **vroot**
- line 11 contains a regex matching all the requests, it is at the bottom because in the routes, order matters.

The contents of the **my_api** contain python modules (API Resources) for example the content of module zun_default/zun_default.py is:

```python
from zunzuncito import tools


class APIResource(object):

    @tools.allow_methods('get, head')
    def dispatch(self, request, response):

        request.log.debug(tools.log_json({
            'API': request.version,
            'URI': request.URI,
            'method': request.method,
            'vroot': request.vroot
        }, True))

        data = {}
        data['about'] = ("Hi %s, I am zunzuncito a micro-framework for creating"
                         " REST API's, you can read more about me in: "
                         "www.zunzun.io") % request.environ.get('REMOTE_ADDR', 0)

        data['Request-ID'] = request.request_id
        data['URI'] = request.URI
        data['Method'] = request.method

        return tools.log_json(data, 4)
```

See also:

Basic template

How to run it

Zunzuncito is compatible with any WSGI server, next are some examples of how to run it with uWSGI, and Gunicorn, Twisted.

uWSGI

Listening on port 8080:

```
uwsgi --http :8080 --wsgi-file app.py --callable app --master
```

Listening on port 80 with 2 processes and stats on http://127.0.0.1:8181:

```
uwsgi --http :80 --wsgi-file app.py --callable app --master --processes 2 --threads 2␣
↪--stats 127.0.0.1:8181 --harakiri 30
```

Using a .ini file

TRACK_ID

route-run adds a custom tracking ID, see uwsgi InternalRouting

```
[uwsgi]
http = :8080
route-run = addvar:TRACK_ID=${uwsgi[uuid]}
route-run = log:TRACK_ID = ${TRACK_ID}
master = true
processes = 2
threads = 1
stats = 127.0.0.1:8181
harakiri = 30
wsgi-file = app.py
callable = app
```

For this case, to append to all your responses the **Request-ID** header run the app like this:

```
app = zunzuncito.ZunZun(root, versions, hosts, routes, rid='TRACK_ID')
```

Gunicorn

Listening on port 8080:

```
gunicorn -b :8080  app:app
```

Listening on port 8080 with 2 processes:

```
gunicorn -b :8080 -w2 app:app
```

GAE

Tu have a ZunZun instance up and running in Google App Engine you can use the following configuration.

Contents of the **app.yaml** file:

main.app

script main is the main.py file **app** is the instance of zunzun

```
application: <your-GAE-application-id>
version: 1
runtime: python27
api_version: 1
threadsafe: yes

handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /.*
  script: main.app
```

Note: When using GAE the global unique identifier per request is: REQUEST_LOG_ID

For this case, to append to all your responses the **Request-ID** header run the app like this:

```
app = zunzuncito.ZunZun(root, versions, hosts, routes, rid='REQUEST_LOG_ID')
```

ZunZun class

ZunZun is the name of the class that will parse all the incoming request and route them to a proper APIResource class to proccess the requests.

```
import zunzuncito

root = 'my_api'

versions = ['v0', 'v1']

hosts = {'*': 'default'}

routes = {'default':[
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST'),
    ('/.*', 'default')
]}

app = zunzuncito.ZunZun(root, versions, hosts, routes, rid='TRACK_ID', debug=True)
```

Root

The `root` argument is the name of the directory containing all your sources.

```
import zunzuncito

root = 'my_api'

versions = ['v0', 'v1']

hosts = {'*': 'default'}

routes = {'default':[
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST'),
    ('/(.*\.(gif|png|jpg|ico|bmp|css|otf|eot|svg|ttf|woff))', 'static')
]}

app = zunzuncito.ZunZun(root, versions, hosts, routes)
```

Making an analogy, you can see `root` as the DocumentRoot of the application.

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
       |--__init__.py
       `--default
          |--__init__.py
          |--v0
          |   |--__init__.py
          |   |--zun_default
          |   |   |--__init__.py
          |   |   `--zun_default.py
          |   `--zun_hasher
          |       |--__init__.py
          |       `--zun_hasher.py
          `--v1
              |--__init__.py
              |--zun_default
              |   |--__init__.py
              |   `--zun_default.py
              `--zun_hasher
                  |--__init__.py
                  `--zun_hasher.py
```

- In this case the **my_api** directory, is the `root`

Versions

The `versions` argument must be a list of names representing the available API versions.

```python
import zunzuncito

root = 'my_api'

versions = ['v0', 'v1']

hosts = {'*': 'default'}

routes = {'default':[
    ('/my/?.*', 'ip_tools', 'GET'),
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST')
]}

app = zunzuncito.ZunZun(root, versions, hosts, routes)
```

Request example

When no version is specified on the URI request, the default version is the first element off the list, example:

```
curl -i http://api.zunzun.io/my/ip
```

Or:

```
curl -i http://api.zunzun.io/v0/my/ip
```

The output could be something like:

```
HTTP/1.1 200 OK
Request-ID: 52ad9da400ff0dda875ef62f7d0001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Sun, 15 Dec 2013 12:16:37 GMT
Server: Google Frontend
Cache-Control: private
```

```
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

{
    "ip": "89.181.199.57"
}
```

Now if we change the version, notice the `v1`:

```
curl -i http://api.zunzun.io/v1/my/ip
```

The output could be something like:

```
HTTP/1.1 200 OK
Request-ID: 52ada62f00ff06ee2d1086b0d00001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Sun, 15 Dec 2013 12:53:03 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

{
"inet_ntoa": 1505085241,
"ip": "89.181.199.57"
}
```

How it works internally

The API directory structre looks like:

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
       |--__init__.py
       `--default
          |--__init__.py
          |--v0
          |  |--__init__.py
          |  |--zun_default
          |  |  |--__init__.py
          |  |  `--zun_default.py
          |  `--zun_ip_tools
          |     |--__init__.py
          |     `--zun_ip_tools.py
          `--v1
             |--__init__.py
             |--zun_default
             |  |--__init__.py
             |  `--zun_default.py
             `--zun_ip_tools
                |--__init__.py
                `--zun_ip_tools.py
```

The directories `v0` and `v1` have the same structure, but the contents of the .py scripts change.

In this example, `v0` returns the IP of the request, while `v1` besides returing the IP it also returns the inet_atom

Hosts

The `hosts` argument contains a dictionary of domains and vroots.

A very basic API, contents of file **app.py** can be:

Hosts dictionary elements

***** wildcard matching all HTTP_HOSTS

default vroot

```python
import zunzuncito

root = 'my_api'

versions = ['v0', 'v1']

hosts = {'*': 'default'}

routes = {'default':[
    ('/my/?.*', 'ip_tools', 'GET')
]}

app = zunzuncito.ZunZun(root, versions, hosts, routes, debug=True)
```

To support multi-tenancy the **hosts** dictionary is needed.

A dictionary structure is formed by **key: value** elements, in this case the key is used for specifying the 'host' and the value to specify the **vroot**

Hosts structure & vroot

The wildcard character * can be used, for example:

```
1 hosts = {
2     '*': 'default',
3     '*.zunzun.io': 'default',
4     'ejemplo.org': 'ejemplo_org',
5     'api.ejemplo.org': 'api_ejemplo_org'
6 }
```

- line 2 matches any host `*` and will be served on vroot '**default**'
- line 3 matches any host ending with `zunzun.io` and will be served on vroot '**default**'
- line 4 matches host `ejemplo.org` and will be server on vroot '**ejemplo_org**'
- line 5 matches host `api.ejemplo.org` and will be served on vroot '**api_ejemplo_org**'

Notice that the vroot values use _ as separator instead of a dot, this is to prevent conflicts on how python read files. for example this request:

```
http://api.ejemplo.org/v0/gevent
```

Internally will be calling something like:

```
import my_api.api_ejemplo_org.v0.zun_gevent.zun_gevent
```

Directory structure

The API directory structure for this example would be:

virtual roots (vroot)

> **default** *line 2, 3*
>
> **ejemplo_org** *line 4*
>
> **api_ejemplo_org** *line 5*

```
/home/
 `--zunzun/
    |--app.py
    `--my_api
       |--__init__.py
       |--default
       |  |--__init__.py
       |  `--v0
       |     |--__init__.py
       |     `--zun_default
       |        |--__init__.py
       |        `--zun_default.py
       |--ejemplo_org
       |  |--__init__.py
       |  `--v0
       |     |--__init__.py
       |     `--zun_default
       |        |--__init__.py
       |        `--zun_default.py
       `--api_ejemplo_org
          |--__init__.py
          `--v0
             |--__init__.py
             |--zun_gevent
             |  |--__init__.py
             |  `--zun_gevent.py
             `--zun_default
                |--__init__.py
                `--zun_default.py
```

Routes

The `routes` argument must be a dictionary containing defined routes per **vroot**.

```
1  import zunzuncito
2
3  root = 'my_api'
4
```

```python
versions = ['v0', 'v1']

hosts = {
    '*': 'default',
    'domain.tld': 'default',
    '*.domain.tld': 'default',
    'beta.domain.tld': 'beta'
}

routes = {'default':[
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST')
],'beta':[
    ('/upload/?.*', 'upload', 'PUT, POST'),
    ('/.*', 'default')
]}

app = zunzuncito.ZunZun(root, versions, hosts, routes, debug=True)
```

Note: By default, if no **routes** specified, the requests are handled by matching the URI request with an valid **API Resource**, you only need to specify **routes** if want to handle different URI requests with a single **API Resource**

Example

The request:

```
http://api.zunzun.io/v0/env
```

Will be handled by the python custom module zun_env/zun_env.py

But all the following GET requests:

- http://api.zunzun.io/v0/md5/freebsd
- http://api.zunzun.io/v0/sha1/freebsd
- http://api.zunzun.io/v0/sha256/freebsd
- http://api.zunzun.io/v0/sha512/freebsd

And also this POST requests:

```
curl -i -X POST http://api.zunzun.io/v0/md5 -d 'freebsd'

curl -i -X POST http://api.zunzun.io/v0/sha1 -d 'freebsd'

curl -i -X POST http://api.zunzun.io/v0/sha256 -d 'freebsd'

curl -i -X POST http://api.zunzun.io/v0/sha512 -d 'freebsd'
```

Will be handled by the pythom custom module zun_hasher/zun_hahser.py, this is because a specified route:

```
('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST')
```

You can totally omit routes and handle all by following the API *directory structure*, this can give you more fine control over you API, for example in the previous example you could create modules for every hash algorithm, and have independent modules like:

```
`--v0
    |--__init__.py
    |--zun_md5
    |   |--__init__.py
    |   `--zun_md5.py
    |--zun_sha1
    |   |--__init__.py
    |   `--zun_sha1.py
    |--zun_sha256
    |   |--__init__.py
    |   `--zun_sha256.py
    `--zun_sha512
        |--__init__.py
        `--zun_sha512.py
```

Note: Defining routes or using the directory structure is a design choice, some times having all in one module can be easy to maintain, while other times having a module for each specific task, would be prefered.

See also:

The zun_ prefix

The flow

When a new request arrive, the ZunZun router searches for a `vroot` declared on the hosts dictionary matching the current HTTP_HOST.

Once a `vroot` is found, the ZunZun router parses the REQUEST_URI in order to accomplish this pattern:

```
/version/api_resource/path
```

The router first analyses the URI and determines if it is versioned or not by finding a match with the current specified versions in case no one is found, fallback to the default which is always the first item on the versions list in case one provided, or `v0`.

After this process, the REQUEST_URI becomes a list of resources - something like:

```
['version', 'api_resource', 'path']

# for  http://api.zunzun.io/v0/env
['v0', 'env']

# for http://api.zunzun.io/v0/sha256/freebsd
['v0', 'sha256', 'freebsd']
```

The second step on the router is to find a match within the `routes` dictionary and the local modules.

In case a list of `routes` is passed as an argument to the ZunZun instance, the router will try to match the API_resource with the items of the `routes` dictionary. If no matches are found it will try to find the module in the root directory.

Routes dictionary structure

In the above example, the `routes` dictionary contains:

vroot	regular expression	API Resource	HTTP methods
default	/(md5|sha1|sha256|sha512)(/.*)?	hasher	'GET, POST'
beta	/upload/?.*	upload	'PUT, POST'
beta	/.*	default	

Translating the table to code:

```
routes = {}
routes['default'] = [
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST')
]
routes['beta'] = [
    ('/upload/?.*', 'upload', 'PUT, POST'),
    ('/.*', 'default')
]
```

Warning: Regular expressions have priority, for example the regex `(/.*)` will **catch-all** the request, that's why in our example is the last regex, since order is important.

Directory structure

The API directory structure for the examples presented here is:

API directory structure

default vroot directory

beta vroot directory

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
        |--__init__.py
        |--default
        |   |--__init__.py
        |   |--v0
        |   |   |--__init__.py
        |   |   |--zun_default
        |   |   |   |--__init__.py
        |   |   |   `--zun_default.py
        |   |   |--zun_env
        |   |   |   |--__init__.py
        |   |   |   `--zun_env.py
        |   |   `--zun_hasher
        |   |       |--__init__.py
        |   |       `--zun_hasher.py
        |   `--v1
        |       |--__init__.py
        |       |--zun_default
        |       |   |--__init__.py
        |       |   `--zun_default.py
        |       `--zun_hasher
        |        |--__init__.py
        |        `--zun_hasher.py
        `--beta
           |--__init__.py
           `--v0
              |--__init__.py
              |--zun_default
              |  |--__init__.py
              |  `--zun_default.py
              `--zun_upload
                 |--__init__.py
                 `--zun_upload.py
```

Prefix

The `prefix` argument is the string that should be appended to all the names of the python modules.

```python
import zunzuncito

root = 'my_api'

versions = ['v0', 'v1']

hosts = {'*': 'default'}

routes = {'default':[
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST')
]}

app = zunzuncito.ZunZun(root, versions, hosts, routes, prefix='zzz_')
```

Note: The default prefix is `zun_`

Directory structure

The directory containing the sources for the application would look like:

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
        |--__init__.py
        `--default
           |--__init__.py
           |--v0
           |  |--__init__.py
           |  |--zzz_default
           |  |  |--__init__.py
           |  |  `--zzz_default.py
           |  `--zzz_hasher
           |     |--__init__.py
           |     `--zzz_hasher.py
           `--v1
```

```
              |--__init__.py
              |--zzz_default
              |   |--__init__.py
              |   `--zzz_default.py
              `--zzz_hasher
                 |--__init__.py
                 `--zzz_hasher.py
```

- In this case the **my_api** directory, is the `root` and all modules (API Resources) start with **zzz_**

Note: The idea of the **prefix** is to avoid conflics with current python modules

See also:

pep 395, python import

Rid

The `rid` argument, contains the name of the environ variable containing the request id if any, for example when using GAE:

```
app = zunzuncito.ZunZun(root, versions, hosts, routes, rid='REQUEST_LOG_ID')
```

This helps to add a Request-ID header to all your responses, example when you make a request like:

```
curl -i http://api.zunzun.io/md5/python
```

The response is something like:

```
HTTP/1.1 200 OK
Request-ID: 52b041e500ff018603acd9c1c60001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Tue, 17 Dec 2013 12:21:57 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

{
"hash": "23eeeb4347bdd26bfc6b7ee9a3b755dd",
"string": "python",
"type": "md5"
}
```

This can help to trace the request in both server/client side.

If you do not specify the `rid` argument the **Request-ID** will automatically be generated using an UUID , so for example you can run the app like this:

```
app = zunzuncito.ZunZun(root, versions, hosts, routes)
# notice there is no rid argument
```

And the output will be something like:

```
HTTP/1.1 200 OK
Request-ID: 44f237e7-a330-4fb3-ba61-be5abd960688
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Tue, 17 Dec 2013 12:21:57 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

{
"hash": "23eeeb4347bdd26bfc6b7ee9a3b755dd",
"string": "python",
"type": "md5"
}
```

Debug

The python logging module is used for creating logs

To enable debugging, set the `debug` argument to **True**, this will log how the API is handling the requests, besides setting the loglevel to **DEBUG** you can run the app like this:

```
app = zunzuncito.ZunZun(root, versions, hosts, routes, debug=True)
```

Note: The default loglevel is **INFO**

Request class

When receiving a request a **request object** is created and passed as an argument to the dispatch method of the APIResource class

The first argument for the dispatch method is the request object:

```
from zunzuncito import tools

class APIResource(object):


    def dispatch(self, request, response):
        """ your code goes here """
```

Request object contents

Name	Description
log	logger intance
request_id	The request id
environ	The wsgi environ
URI	REQUEST_URI or PATH_INFO
host	The host name.
method	The request method (GET, POST, HEAD, etc)
path	list of URI elements
resource	Name of the API resource
version	Current version
vroot	Name of the vroot

Example

```
from urlparse import parse_qsl
from zunzuncito import tools

class APIResource(object):


    @tools.allow_methods('get, post')
    def dispatch(self, request, response):

        """
        log this request
        """
        request.log.info(tools.log_json({
            'API': request.version,
            'Method': request.method,
            'URI': request.URI,
            'vroot': request.vroot
        }, True))

        if request.method == 'POST':
            data = dict(parse_qsl(request.environ['wsgi.input'].read(), True))
        else
            data = dict(parse_qsl(request.environ['QUERY_STRING'], True))

        data = {k: v.decode('utf-8') for k, v in data.items()}

        return tools.log_json(data)
```

Response class

All requests need a response, the `response` class creates an object for every request, the one can be used to send custom headers or HTTP status codes.

The second argument for the dispatch method is the response object:

```
from zunzuncito import tools

```

```
class APIResource(object):


    def dispatch(self, request, response):
        """ your code goes here """
```

Response object contents

Name	Description
log	logger intance.
request_id	The request id.
headers	A CaseInsensitiveDict instance, for storing the headers.
status	Default **200** an int respresenting an HTTP status code.
start_response	The start_response() Callable.

Example

```
from zunzuncito import tools


class APIResource(object):

    def __init__(self):
        self.headers['Content-Type'] = 'text/html; charset=UTF-8'

    def dispatch(self, request, response):

        response.headers.update(self.headers)

        try:
            name = request.path[0]
        except Exception:
            name = ''

       if name:
             return 'Name: ' + name

        response.status =  406
```

API Resource

APIResource class

`APIResource` is the name of the class that the ZunZun instance will call to handle the incoming requests.

APIResource.dispatch

The ZunZun instance always will call the `dispath` method that belongs to the APIResource class.

```python
from zunzuncito import tools


class APIResource(object):

   def dispatch(self, request, response):

        request.log.debug(tools.log_json({
            'API': request.version,
            'URI': request.URI,
            'method': request.method,
            'vroot': request.vroot
        }, True))

        # print all the environ
        return tools.log_json(request.environ, 4)
```

For example, the following request:

```
http://127.0.0.1:8080/v0/upload
```

Is handled by the custom python module `zun_upload/zun_upload.py` which contents:

```python
"""
upload resource

Upload by chunks

@see http://www.grid.net.ru/nginx/resumable_uploads.en.html
"""
import os
from zunzuncito import tools


class APIResource(object):

    @tools.allow_methods('post, put')
    def dispatch(self, request, response):
        try:
            temp_name = request.path[0]
        except:
            raise tools.HTTPException(400)

        """rfc2616-sec14.html
        see http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
        see http://www.grid.net.ru/nginx/resumable_uploads.en.html
        """
        content_range = request.environ.get('HTTP_CONTENT_RANGE', 0)

        length = int(request.environ.get('CONTENT_LENGTH', 0))

        if content_range:
            content_range = content_range.split()[1].split('/')

            index, offset = [int(x) for x in content_range[0].split('-')]

            total_size = int(content_range[1])

            if length:
                chunk_size = length
            elif offset > index:
                chunk_size = (offset - index) + 1
            elif total_size:
                chunk_size = total_size
            else:
                raise tools.HTTPException(416)
        elif length:
            chunk_size = total_size = length
            index = 0
            offset = 0
        else:
            raise tools.HTTPException(400)

        stream = request.environ['wsgi.input']

        body = []

        try:
            temp_file = os.path.join(
                os.path.dirname('/tmp/test_upload/'),
                temp_name)

            with open(temp_file, 'a+b') as f:
                original_file_size = f.tell()

                f.seek(index)
                f.truncate()

                bytes_to_write = chunk_size

                while chunk_size > 0:
                    # buffer size
                    chunk = stream.read(min(chunk_size, 1 << 13))
                    if not chunk:
                        break
                    f.write(chunk)
                    chunk_size -= len(chunk)

                f.flush()
                bytes_written = f.tell() - index

                if bytes_written != bytes_to_write:
                    f.truncate(original_file_size)
                    f.close()
                    raise tools.HTTPException(416)

            if os.stat(temp_file).st_size == total_size:
                response.status = 200
            else:
                response.status = 201
                body.append('%d-%d/%d' % (index, offset, total_size))

            request.log.info(tools.log_json({
                'index': index,
                'offset': offset,
                'size': total_size,
                'status': response.status,
                'temp_file': temp_file
            }, True))

            return body
        except IOError:
            raise tools.HTTPException(
                500,
                title="upload directory [ %s ]doesn't exist" % temp_file,
                display=True)
```

Note: All the custom modules must have the **APIResource** class and the method **dispatch** in order to work

dispatch method

The `dispatch` method belongs to the APIResource class and is called by the ZunZun instance in order to process the requests.

Basic template

```
from zunzuncito import tools

class APIResource(object):


    def dispatch(self, request, response):
        """ your code goes here """
```

Status codes

The default HTTP status code is 200, but based on your needs you can change it to fit you response very eazy by just doing something like:

```
response.status = 201
```

Headers

As with the status codes, same happens with the HTTP headers, The default headers are:

```
Content-Type: 'application/json; charset=UTF-8'
Request-ID: <request_id>
```

For updating/replacing you just need to do something like:

```
response.headers['my_custom_header'] = str(uuid.uuid4())
```

Example

```python
from zunzuncito import tools

class APIResource(object):

    def __init__(self, api):
        self.headers = {'Content-Type': 'text/html; charset=UTF-8'}

    def dispatch(self, request, response):

        try:
            name = self.api.path[0]
        except:
            name = ''

        if name:
            response.headers['my_custom_header'] = name
        else:
            response.status = 406

        response.headers.update(self.headers)

        return 'Name: ' + name
```

The output for:

```
curl -i http://api.zunzun.io/status_and_headers
```

```
HTTP/1.1 406 Not Acceptable
Request-ID: 52e78a1500ff0f217359e91eb90001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Tue, 28 Jan 2014 10:44:38 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

Name:
```

The output for:

```
curl -i http://api.zunzun.io/status_and_headers/foo
```

```
HTTP/1.1 200 OK
Request-ID: 52e78a9300ff0f3fe44a7e4fbf0001737e7a756e7a756e6369746f2d617069000131000100
my_custom_header: foo
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Tue, 28 Jan 2014 10:46:44 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

Name: foo
```

See also:

pep 0333

@allow_methods decorator

The `@allow_methods` decorator when applied to the dispatch method works like a filter to specified HTTP methods

Example

```
from zunzuncito import tools

class APIResource(object):

    @tools.allow_methods('post, put')
    def dispatch(self, request, response):
        """ your code goes here """
```

In this case all the request that are not **POST** or **PUT** will be rejected

path

`path` is the name of the variable containing a list of elements of the URI after has been parsed.

Suppose the incoming request is:

```
http://api.zunzun.io/v1/gevent/ip
```

ZunZun instance will convert it to:

```
['v1', 'gevent', 'ip']
```

where:

```
vroot = default
version = v1
resource = gevent
path = ['ip']
```

for the incoming request:

```
http://api.zunzun.io/gevent/aa/bb
```

this will be generated:

```
vroot = default
version = v0
resource = gevent
path = ['aa', 'bb']
```

Example

```python
from zunzuncito import tools


class APIResource(object):

    def dispatch(self, request, response):

        try:
            name = request.path[0]
        except Exception:
            name = ''

        return 'Name: ' + name
```

URI module

Every resource has a module, this is made with the intention to have more order and give more flexibility.

URI parts

```
http://api.zunzun.io/v0/get/client/ip
\__________________/\_/\__/\_____/\_/
         |           |   | \__|____|/
         |       version |    |  | |
   host (default)    resource |path|
                              |    |
                           path[0] |
                                   |
                                path[1]
```

ZunZun translates that URI to:

```
my_api.default.v0.zun_get.zun_client.zun_client
```

See also:

URI scheme, Uniform_resource_locator

_catchall resource

ZunZun process request only for does who have an existing module, for example using the following directory structure notice we only have 2 modules, `zun_default`, `zun_hassher` and `zun_gevent`.

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
        |--__init__.py
        `--default
           |--__init__.py
           `--v0
              |--__init__.py
              |--zun_default
```

```
            |   |--__init__.py
            |   `--zun_default.py
            |--zun_hasher
            |   |--__init__.py
            |   `--zun_hasher.py
            `--zun_gevent
                |--__init__.py
                `--zun_gevent.py
```

The routes uses regular expresions to match more then one request into one module, for example:

```
routes = {'default':[
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST'),
    ('/.*', 'notfound')
]}
```

For example the `zun_hasher` module will handle all the request for URI containing:

```
/md5, /sha1, /sha256, /sha512
```

But the regex:

```
'/.*'
```

Will match anything and send it to the `zun_notfound` module, the problem with this regex, is that since it will catch anything, if you make a request for example to:

```
api.zunzun.io/gevent
```

It will be processed by the `zun_notfound` since the regex catched the request.

A solution to this, could be to add a route for the gevent request, something like this:

```
routes = {'default':[
    ('/(md5|sha1|sha256|sha512)(/.*)?', 'hasher', 'GET, POST'),
    ('/gevent/?.*', 'gevent'),
    ('/.*', 'notfound')
]}
```

That could solve the problem, but forces you to have a route for every module, the more modules you have, the more complicated and dificult becomes to maintain the routes dictionary.

So, be available to have regular expresions mathing many to one module, to continue serving directly from modules that don't need a regex, and to also have a catchall that does not need require a regex and it is olny called when all the routes and modules options have been exhauste, the **_catchall** module was created.

How it works

The only thing required, is to create a module with the name `__catchall`, if using the default prefix, it would be `zun__catchall`.

- Notice the double `__catchall` underscore.

See also:

The zun prefix

Directory structure

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
        |--__init__.py
        `--default
           |--__init__.py
           `--v0
              |--__init__.py
              |--zun_default
              |  |--__init__.py
              |  `--zun_default.py
              |--zun_hasher
              |  |--__init__.py
              |  `--zun_hasher.py
              |--zun_gevent
              |  |--__init__.py
              |  `--zun_gevent.py
              `--zun__catchall
                 |--__init__.py
                 `--zun__catchall.py
```

When processing a request, if not module is found either in the routes or in the directory structure, if the If the `__catchall` module is found, it is goint to be used for handling the request, if not it will just return an HTTP 501 Not Implementd status.

Example

The following example, will handle all *not found* modules for the incoming request, and redirect to '/'.

```python
"""
catchall resource
"""

from zunzuncito import tools


class APIResource(object):

    def __init__(self):
        self.headers = {'Content-Type': 'text/html; charset=UTF-8'}

    def dispatch(self, request, response):
        request.log.debug(tools.log_json({
            'API': request.version,
            'URI': request.URI,
            'rid': request.request_id,
            'vroot': request.vroot
        }, True))

        response.headers.update(self.headers)

        raise tools.HTTPException(302, headers={'Location': '/'}, log=False)
```

tools

tools is a module that containg a set of classes and functions that help to proccess the reply of the request more easy.

```
from zunzuncito import tools

class APIResource(object):

    def dispatch(self, request, response):
        """ your code goes here """
```

HTTPException

The `HTTPException` class extends the HTTPError class, the main idea of it, is to handle posible erros and properly reply with the corresponding HTTP status code.

```
HTTPException(status, title=None, description=None, headers=None, code=None,␣
↪display=False, log=True)
```

Note: Description argument must be a dictionary containing HTTP header fields

Redirect example

```
from zunzuncito import tools


class APIResource(object):

    def dispatch(self, request, response):
        try:
            name = request.path[0]
        except:
            raise tools.HTTPException(
                302,
                headers={'Location': '/home'},
                log=False}
```

Bad request example

```
from zunzuncito import tools


class APIResource(object):

    def dispatch(self, request, response):
        try:
            name = request.path[0]
        except:
            raise tools.HTTPException(400)
```

the line:

```
raise tools.HTTPException(400)
```

For a request like:

```
curl -i http://api.zunzun.io/exception
```

Will reply with this:

```
HTTP/1.1 400 Bad Request
Request-ID:
52c597a700ff0229fef9f477280001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Date: Thu, 02 Jan 2014 16:45:27 GMT
Server: Google Frontend
Content-Length: 0
Alternate-Protocol: 80:quic,80:quic
```

This is because the request URI is missing the path and should be something like:

```
curl -i http://api.zunzun.io/exception/foo
```

That will return something like:

```
HTTP/1.1 200 OK
Request-ID:
52c597d200ff0d89f81dcec4280001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Thu, 02 Jan 2014 16:46:11 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

my_api.default.v0.zun_exception.zun_exception
```

Note: If you only pass the integer HTTP status code to the HTTPExecption, only the response headers will be sent.

Body response

Besides only replying with the headers you may want to give a more informative / verbose message, the HTTPExeption accept the following arguments:

```
HTTPException(status, title=None, description=None, headers=None, code=None,␣
↪display=False, log=True)
```

For example the following snippet of code taken from zun_exception.py:

```
 if name != 'foo':
     raise tools.HTTPException(
         406,
         title='exeption example',
         description='name must be foo',
         code='my-custom-code',
         display=True)
```

When the request is:

```
curl -i http://api.zunzun.io/v0/exception/naranjas
```

Notice that the path in this case is:

```
path = ['naranjas']
```

Will reply with something like:

```
HTTP/1.1 406 Not Acceptable
Request-ID:␣
↪52c59bdf00ff0b7042cbfd5d120001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Vary: Accept-Encoding
Date: Thu, 02 Jan 2014 17:03:27 GMT
Server: Google Frontend
Cache-Control: private
Alternate-Protocol: 80:quic,80:quic
Transfer-Encoding: chunked

{
    "code": "my-custom-code",
    "description": "name must be foo",
    "status": "406",
    "title": "exeption example"
}
```

MethodException

While defining routes or using the @allow_methods decorator you can specify the allowed HTTP methods to support, the ZunZun instance, internally will verify for the corret method, otherwise will raise an `MethodException`

The `MethodException` behaves similar to the HTTPException the only difference is that by default sets the status code to 405 Method Not Allowed

```
MethodException(status=405, title=None, description=None, headers=None, code=None,␣
↪display=False)
```

For example the zun_exception.py custom module only accepts 'GET' methods from this code snippet:

```
@tools.allow_methods('get')
def dispatch(self, request, response):
```

Therefor if you try the following:

```
curl -i -X HEAD http://api.zunzun.io/exception/foo
```

- Note the: **-X HEAD** this will send a **HEAD** request not a **GET**

The answer will be simillar to:

```
HTTP/1.1 405 Method Not Allowed
Request-ID: 52c6ac4e00ff060346c67c66450001737e7a756e7a756e6369746f2d617069000131000100
Content-Type: application/json; charset=UTF-8
Date: Fri, 03 Jan 2014 12:25:50 GMT
Server: Google Frontend
Content-Length: 0
Alternate-Protocol: 80:quic,80:quic
```

allow_methods

The `allow_methods` is a function used as decorator

See also:

@allow_methods decorator

log_json

The `log_json` is a function that given a dictionary, returns a json structure.

This helps that logs can be parsed and processed by external tools.

The arguments are:

```
log_json(log, indent=False)
```

log a python dictionary

indent returns the json structured indented (more human readable)

Note: If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, or negative, will only insert newlines. None (the default) selects the most compact representation.

See also:

python json

Example

```python
from zunzuncito import tools

class APIResource(object):

    def dispatch(self, request, response):

        request.log.debug(tools.log_json({
            'API': request.version,
            'Method': request.method,
            'URI': request.URI,
            'vroot': request.vroot
        }, True))

        """ your code goes here """
```

See also:

Structured Logging

clean_dict

The `clean_dict` is a small recursive function that given a dictionary will try to 'clean it' converting it to a string:

```
clean_dict(dictionary)
```

It is commonly used in conjunction with the log_json function.

Example

snippet taken from zun_self.py:

```
def dispatch(self, request, response):

    return (
        json.dumps(
            tools.clean_dict(request.__dict__),
            sort_keys=True,
            indent=4)
    )
    # taking advantage of the tools.log_json
    # return tools.log_json(request.__dict__, 4)
```

CaseInsensitiveDict

A case-insensitive `dict`-like object.

Note: Class taken from: requests

Implements all methods and operations of `collections.MutableMapping` as well as dict's `copy`. Also provides `lower_items`.

All keys are expected to be strings. The structure remembers the case of the last key to be set, and `iter(instance)`, `keys()`, `items()`, `iterkeys()`, and `iteritems()` will contain case-sensitive keys. However, querying and contains testing is case insensitive:

```
cid = CaseInsensitiveDict()
cid['Accept'] = 'application/json'
cid['aCCEPT'] == 'application/json'  # True
list(cid) == ['Accept']  # True
```

For example, `headers['content-encoding']` will return the value of a `'Content-Encoding'` response header, regardless of how the header name was originally stored.

If the constructor, `.update`, or equality comparison operations are given keys that have equal `.lower()` s, the behavior is undefined.

HTTP status codes

```
# 1xx: Informational - Request received, continuing process
HTTP_100 = '100 Continue' # [RFC2616]
HTTP_101 = '101 Switching Protocols' # [RFC2616]
HTTP_102 = '102 Processing' # [RFC2518]

# 2xx: Success - The action was successfully received, understood, and accepted
HTTP_200 = '200 OK' # [RFC2616]
HTTP_201 = '201 Created' # [RFC2616]
HTTP_202 = '202 Accepted' # [RFC2616]
HTTP_203 = '203 Non-Authoritative Information' # [RFC2616]
HTTP_204 = '204 No Content' # [RFC2616]
HTTP_205 = '205 Reset Content' # [RFC2616]
HTTP_206 = '206 Partial Content' # [RFC2616]
HTTP_207 = '207 Multi-Status' # [RFC4918]
HTTP_208 = '208 Already Reported' # [RFC5842]
HTTP_226 = '226 IM Used' # [RFC3229]

# 3xx: Redirection - Further action must be taken in order to complete the request
HTTP_300 = '300 Multiple Choices' # [RFC2616]
HTTP_301 = '301 Moved Permanently' # [RFC2616]
HTTP_302 = '302 Found' # [RFC2616]
HTTP_303 = '303 See Other' # [RFC2616]
HTTP_304 = '304 Not Modified' # [RFC2616]
HTTP_305 = '305 Use Proxy' # [RFC2616]
HTTP_306 = '306 Reserved' # [RFC2616]
HTTP_307 = '307 Temporary Redirect' # [RFC2616]
HTTP_308 = '308 Permanent Redirect' # [RFC-reschke-http-status-308-07]

# 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
HTTP_400 = '400 Bad Request' # [RFC2616]
HTTP_401 = '401 Unauthorized' # [RFC2616]
HTTP_402 = '402 Payment Required' # [RFC2616]
HTTP_403 = '403 Forbidden' # [RFC2616]
HTTP_404 = '404 Not Found' # [RFC2616]
HTTP_405 = '405 Method Not Allowed' # [RFC2616]
HTTP_406 = '406 Not Acceptable' # [RFC2616]
HTTP_407 = '407 Proxy Authentication Required' # [RFC2616]
HTTP_408 = '408 Request Timeout' # [RFC2616]
HTTP_409 = '409 Conflict' # [RFC2616]
HTTP_410 = '410 Gone' # [RFC2616]
HTTP_411 = '411 Length Required' # [RFC2616]
HTTP_412 = '412 Precondition Failed' # [RFC2616]
HTTP_413 = '413 Request Entity Too Large' # [RFC2616]
HTTP_414 = '414 Request-URI Too Long' # [RFC2616]
HTTP_415 = '415 Unsupported Media Type' # [RFC2616]
HTTP_416 = '416 Requested Range Not Satisfiable' # [RFC2616]
HTTP_417 = '417 Expectation Failed' # [RFC2616]
HTTP_422 = '422 Unprocessable Entity' # [RFC4918]
HTTP_423 = '423 Locked' # [RFC4918]
HTTP_424 = '424 Failed Dependency' # [RFC4918]
HTTP_425 = '425 Unassigned'
HTTP_426 = '426 Upgrade Required' # [RFC2817]
HTTP_427 = '427 Unassigned'
HTTP_428 = '428 Precondition Required' # [RFC6585]
HTTP_429 = '429 Too Many Requests' # [RFC6585]
HTTP_430 = '430 Unassigned'
```

```
HTTP_431 = '431 Request Header Fields Too Large' # [RFC6585]

# 5xx: Server Error - The server failed to fulfill an apparently valid request
HTTP_500 = '500 Internal Server Error' # [RFC2616]
HTTP_501 = '501 Not Implemented' # [RFC2616]
HTTP_502 = '502 Bad Gateway' # [RFC2616]
HTTP_503 = '503 Service Unavailable' # [RFC2616]
HTTP_504 = '504 Gateway Timeout' # [RFC2616]
HTTP_505 = '505 HTTP Version Not Supported' # [RFC2616]
HTTP_506 = '506 Variant Also Negotiates (Experimental)' # [RFC2295]
HTTP_507 = '507 Insufficient Storage' # [RFC4918]
HTTP_508 = '508 Loop Detected' # [RFC5842]
HTTP_509 = '509 Unassigned'
HTTP_510 = '510 Not Extended' # [RFC2774]
HTTP_511 = '511 Network Authentication Required' # [RFC6585]
```

See also:

HTTP status codes

zunzuncito Package

`zunzuncito` Package

`http_status_codes` Module

`request` Module

`response` Module

`tools` Module

`zunzun` Module

WebOb

What is it?

WebOb is a Python library that provides wrappers around the WSGI request environment, and an object to help create WSGI responses. The objects map much of the specified behavior of HTTP, including header parsing, content negotiation and correct handling of conditional and range requests.

This helps you create rich applications and valid middleware without knowing all the complexities of WSGI and HTTP.

Why ?

The ZunZun instance allows you to handle the request by following defined routes and by calling the dispatch method, the way you process the request is up to you, you can either do all by your self or use tools that can allow you to simplify this process, for this last one, **WebOb** is a library that integrates very easy and that may help you to parse the GET, POST arguments, set/get cookies, etc; with out hassle.

Example

The following code, handles the request for http://api.zunzun.io/webob.

```python
from webob import Request
from zunzuncito import tools


class APIResource(object):


    def dispatch(self, request, response):
        req = Request(request.environ)

        data = {}
        data['req-GET'] = req.GET
        data['req-POST'] = req.POST
        data['req-application_url'] = req.application_url
        data['req-body'] = req.body
        data['req-content_type'] = req.content_type
        data['req-cookies'] = req.cookies
        data['req-method'] = req.method
        data['req-params'] = req.params
        data['req-path'] = req.path
        data['req-path_info'] = req.path_info
        data['req-path_qs'] = req.path_qs
        data['req-path_url'] = req.path_url
        data['req-query_string'] = req.query_string
        data['req-script_name'] = req.script_name
        data['req-url'] = req.url

        return tools.log_json(data, 4)
```

Basically you only need to pass the **environ** argument to the `webob.Request`:

```python
def dispatch(self, environ):
    req = Request(environ)
    """ your code goes here """
```

See also:

WebOb Request

GAE

When using google app engine you need to add this lines to your app.yaml file in order to be available to import webob:

```yaml
libraries:
- name: webob
  version: latest
```

Jinja2

Jinja2 is a modern and designer friendly templating language for Python, modelled after Django's templates. It is fast, widely used and secure with the optional sandboxed template execution environment:

```
<title>{% block title %}{% endblock %}</title>
<ul>
{% for user in users %}
    <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

Why ?

Zunzuncito was made mainly for responding in json format not HTML, but also one of the design goals is to give the ability to create almost anything easy, therefor if you need to display HTML, Jinja2 integrates very easy.

Example

The following code, handles the request for: http://api.zunzun.io/jinja2.

```
import os

from jinja2 import Environment, FileSystemLoader
from zunzuncito import tools

jinja = Environment(autoescape=True, loader=FileSystemLoader(
    os.path.join(os.path.abspath(os.path.dirname(__file__)), 'templates')))


class APIResource(object):

    def __init__(self):
        self.headers['Content-Type'] = 'text/html; charset=UTF-8'

    def dispatch(self, request, response):

        request.log.debug(tools.log_json({
            'API': request.version,
            'method': request.method,
            'URI': request.URI,
            'vroot': request.vroot
        }, True))

        response.headers.update(self.headers)

        template_values = {
            'IP': request.environ.get('REMOTE_ADDR', 0)
        }

        template = jinja.get_template('example.html')

        return template.render(template_values).encode('utf-8')
```

The example.html contains:

```
<html>
    <head>
        <meta charset="utf-8">
        <title>{{ IP }}</title>
    </head>

    <body>
    <h3>IP: {{ IP }}</h3>
    </body>
</html>
```

Directory structure

```
/home/
  `--zunzun/
     |--app.py
     `--my_api
        |--__init__.py
        `--default
           |--__init__.py
           `--v0
              |--__init__.py
              `--zun_jinja2
                 |--__init__.py
                 |--zun_jinja2.py
                 `--templates
                    `--example.html
```

See also:

zun_jinja2 API resource & Jinja2 Basics

GAE

When using google app engine you need to add this lines to your app.yaml file in order to be available to import jinja2:

```
libraries:
- name: jinja2
  version: latest
```

gevent

gevent is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libev event loop.

When using gevent and 'yield' you may want to call the 'start_response' before so that it can send your proper headers, for this you must use something like:

```
response.send()
```

See also:

The start_response() Callable

Example

The following code will:

- output the result using content-type: ‘text/html’,
- print ‘sleep 1 second.’ and call gevent.sleep(1),
- print ‘sleep 3 second...’ and call gevent.sleep(3),
- print ‘done. getting some ips...’ call gevent.socket.gethostbyname

An example of the output can be seen here: https://www.youtube.com/watch?v=0N6qXkT-t5E; notice that the prints are secuencial not in one shot.

```
import gevent
import gevent.socket

from zunzuncito import tools


def bg_task():
    for i in range(1, 10):
        print "background task", i
        gevent.sleep(2)


def long_task():
    for i in range(1, 10):
        print i
        gevent.sleep()


class APIResource(object):

    def __init__(self):
        self.headers = {'Content-Type': 'text/html; charset=UTF-8'}

    def dispatch(self, request, response):

        request.log.debug(tools.lo g_json({
            'API': request.version,
            'Method': request.method,
            'URI': request.URI,
            'vroot': request.vroot
        }, True))

        response.headers.update(self.headers)

        """
        calls start_response
        """
        response.send()

        t = gevent.spawn(long_task)
        t.join()

        yield "sleep 1 second.<br/>"

        gevent.sleep(1)

        yield "sleep 3 seconds...<br/>"

        gevent.sleep(3)

        yield "done.<br/>getting some ips...<br/>"

        urls = [
            'www.google.com',
            'www.example.com',
            'www.python.org',
            'zunzun.io']

        jobs = [gevent.spawn(gevent.socket.gethostbyname, url) for url in urls]
        gevent.joinall(jobs, timeout=2)

        for j in jobs:
            yield "ip = %s<br/>" % j.value

        gevent.spawn(bg_task)
```

Changelog

0.1.16 (2014-03-09)

- fixed bug on py_mod to allow sub modules based on the URI to work properly, see URI_module
- fixed __init__ to make custom versions match allowed_URI_chars `^[\w-]+$`
- changed UUID4 to UUID1

0.1.15 (2014-02-27)

- log when trying to load the _catchall, if no _catchall raise Exception about the missing module
- replaced iteritems with items() to be Python 3 compatible

0.1.14 (2014-02-26)

- replaced itertools.ifilter with filter
- improve py_mod if a URI ends with an slash for example: http://api.zunzun.io/v1/add/user/, the py_mod will be: `zun_add/zun_user/zun_user.py`

0.1.13 (2014-02-17)

- Added the log option to the HTTPException, if set to True it will log the exception otherwise not.

0.1.12 (2014-02-13)

- Fixed core to be thread safe.
- New classes request, response, the dispatch method require this `dispatch(self, request, response)`.
- lazy load of resources.
- __catchall module

0.1.11 (2014-02-04)

- Fixed bug in http_status_codes.py when handling generic reasons.

0.1.10 (2014-01-28)

- dispatch method requires now only one argument, which is **environ**, the start_response is handled by the API it self.
- http_status_codes now is a dictionary.

0.1.9 (2014-01-06)

- self._headers is created only once at the beginning and per request just copied to self.headers.

0.1.8 (2014-01-04)

- Fixed tools.log_json function to not indent when no indent value is set.

Issues

Please report any problem, bug, here: https://github.com/nbari/zunzuncito/issues

FAQ

Why I get a warnings for some requests?

If you get something like this:

```
WARNING  2014-03-07 10:01:30,845 zunzun.py:102] {
 "API": "v0",
 "HTTPError": "501",
 "URI": "/",
 "body": {
  "code": "None",
  "description": "No module named myapp_api.default.v0.zun_default.zun_default",
  "display": "False",
  "headers": "None",
  "log": "True",
  "status": "501",
  "title": "ImportError: myapp_api.default.v0.zun_default.zun_default, myapp_api.
↪default.v0.zun__catchall.zun__catchall: No module named myapp_api.default.v0.zun__
↪catchall.zun__catchall"
 },
 "method": "GET",
 "rid":
↪"f8df3ffc8294c0ec0fcc10bbc7c4bfe0febbcaaaf83ff619ab56ca0209225dd0d5f1fd19e42f6b4c1fa2ef0a1f3127
↪"
}
```

Is because you could be missing an `__init__.py`, all the subdirectories of your *API* need need to be treated like python modules.

See also:

This is the directory structure.