OMNI Documentation

Release 1

Adrián Pérez de Castro

April 28, 2016

Contents

Contents:

CHAPTER 1

Configuration

OMNI is configured using a text-based configuration file which is typically located at `/etc/omni.conf`. The configuration file is read using the wcfg module, so it uses the same basic syntax as recognized by it.

1.1 Stores

Each source that can provide OMNI with a method to authenticate users and information about them is a *store*. Multiple stores can be defined in the configuration, and any number of stores can be grouped under a *realm*. Stores can be used for authentication themselves, too.

1.1.1 Plain Text

1.1.2 PAM

Authenticates users using PAM.

Uses PAM (via the simplepam module) to authenticate users using a PAM service. By default, the `login` service will be used, which typically authenticates user accounts for the local machine. You may want to use the `service` option to change this:

```
stores {
    pam.omni {
        service "omni"
    }
}
```

With this snippet in the configuration file, the `/etc/pam.d/omni` service definition file will be expected to be readable by OMNI.

Configuration options:

`service` (optional) Name of the PAM service used to perform authentication. If not provided the `login` service is used by default.

`min-uid` (optional) Users with an UID smaller that this value will not be useable. This is typically used to hide special system users from user listings. The default value is *1000*, which is a typical value for GNU/Linux systems and some BSD systems as well.

1.1.3 Trivial

Authenticates a single user with a fixed password.

An username and password pair is kept in memory to be checked against. A typical usage of this store is to provide a single user that is known only by OMNI; for example, to define an user with administrative privileges that can access the OMNI web interface without restrictions:

```
stores {
    trivial.omni-admin {
        username "baron"
        password "redtriplane"
    }

    # Other stores...
}

http {
    web {
        admin "trivial.omni-admin"
    }
}
```

Configuration options:

`username` Fixed user name.

`password` (optional) Password for the user, in plain text.

1.2 Realms

A *realm* is a collection of *stores*. Authentication and authorization are typically performed by using a realm. When checking credentials, a realm will try each one of the methods from the `method` list, in order. It is enough for one of the methods to succeed; otherwise access is denied if all the methods fail to grant access. Any number of realms can be defined, and an optional description can be attached to them:

```
stores {
        # Configure a couple of stores.
        pam.pam {
                service: "login"
        }
        trivial.simple {
                username: "alice"
                password: "s3cr3t"
        }
}

realms {
        default {
                # Methods are tried in the order defined here.
                methods: ["trivial.simple", "pam.pam"]

                # The description is optional, if missing the name of
                # the realm (in this case, "default") will be used.
                description: "Tries trivial, fall-backs to PAM"
        }
}
```

CHAPTER 2

API Reference

2.1 omni.web.routing

Provides the root resource of the OMNI Web User Interface

2.1.1 Route Template Syntax

Route template strings provide a pattern with wildcards used to match URLs. URL portions matching wildcards are extracted from the URL and converted according to optional type annotations.

A template may not specify any wildcards. In that case the URL is matched literally (e.g. `foo/bar`).

Wildcards are names enclosed in brackets, e.g. `{name}`. A type annotation may optionally follow after a colon, e.g. `{name:int}`. Names must be unique for each route, and they must be valid Python identifiers. Type annotations also determine how values are converted before being passed to the route callback. The following type annotations are available:

Annotation	Type	Converted Type
`id`	Identifier *(default)*	`unicode`
`dotid`	Identifier which may contain dots	`unicode`
`str`	Arbitrary text	`unicode`
`int`	Integer	`int`
`uint`	Unsigned integer	`int`

As an example, the template `{item}/{action}` matches as follows:

Path	Result
/123/save	`{"item": u"123", "action": u"save"}`
/123/save/	*No match.*
/123/	*No match.*
//save	*No match.*

2.1.2 Module Contents

class omni.web.routing.**Route**(*callback*, *template*, *methods*, *name=None*)

Defines a callback to be invoked for a certain URL template.

See *route template syntax* for details on the mini-language accepted in URL template strings.

Parameters

- **callback** – Callable to invoke when the route is matched.
- **template** – Route template string.
- **methods** – HTTP methods handled by this route.
- **name** – Name of the route. May be used to look the route back by name.

make_url(*base=''*, ***kw*)
Builds a matching URL for the route.

This method allows to construct valid URLs that would match the route template. The passed keyword arguments are used to fill-in the wildcards from the template.

Parameters

- **base** – Base URL used as prefix.
- **kw** – Parameter values.

Returns URL as a string.

match_url(*url*)
Try to match an URL against the route template.

Matches the given *url* agains the route template. On a successful match, a dictionary mapping wildcards to the values they matched in the *url* is returned. If the *url* does not match the template, *None* is returned instead.

Parameters **url** – The URL to be matched.

Returns *None* (if unmatched) or *dict* (if matched).

validate_data(*data*)
Validate and convert data according to annotations in the template.

Uses the type annotations from the route template to validate a dictionary in which keys are the names of the wildcards, and their values are strings. A new dictionary is returned, with the values converted accordingly. Typically, this method is used on the data dictionary returned by *match_url()*.

See *route template syntax* for details on how conversions are performed on the input data.

If validation fails `omni.valid.SchemaError` is raised.

Parameters **data** – Dictionary with data.

Returns New dictionary with data validated and converted.

validate_url(*url*)
Matches, validates and convert data from an URL in a single step.

This is equivalent to use *match_url()* followed by *validate_data()*. The returned dictionary will contain values which are already converted.

Parameters **url** – The URL to be matched.

Returns *None* (if unmatched) or *dict* (if matched).

class `omni.web.routing.`**Dispatcher**
Handles dispatching of HTTP requests.

add_route(*route*)
Adds a route to the dispatcher.

Parameters **route** – An instance of *Route*

dispatch_request(*request*)
Dispatches a request to the known routes.

Parameters **request** – A `webob.Request` instance.

dispatch_wsgi(*environ*, *start_response*)
Dispatches a WSGI request.

Parameters

- **environ** – WSGI request environment.
- **start_response** – WSGI response callback

Returns A valid WSGI response content.

plug_routes(*routes*)
Imports all the routes from another dispatcher.

Picks the routes known by *other* and plugs them into the dispatcher

Parameters **routes** – Iterable containing *Route* instances.

url(*route=None*, *base=None*, ***kw*)
Builds an URL for a given route.

Note that *route* is not supplied, the name of the caller will be used as the route name. This is provided as a shorthand for those routes which want to refer to themselves.

If *base* is not supplied, if there is a local variable in the frame of the caller named `request` which is an instance of `webob.Request`, it will be used as if the request was passed as *base*.

Parameters

- **route** – Name of the route, a *Route* instance, or `None`.
- **base** – Base URL, either a string, a `webob.Request` instance from which to infer the base URL, or `None`.
- **kw** – Keyword arguments to fill-in the wildcards in the route URL template.

Returns A string.

routes
Enumerate all the *Route* objects known by the dispatcher.

Accessing this property returns a generator which yields all the routes known by this dispatcher.

omni.web.routing.**route**(*template*, *method='ANY'*, *name=None*)
Decorates a function as a route.

Parameters

- **template** – Route template (see *route template syntax*).
- **method** – HTTP methods for the route, either a single value or a list of strings.
- **name** – Name of the route. If not provided, the name of the decorated function is used.

omni.web.routing.**get**(*template*, **arg*, ***kw*)
Decorates a function as a route, using `GET` as method.

See *route()* for details.

omni.web.routing.**post**(*template*, **arg*, ***kw*)
Decorates a function as a route, using `POST` as method.

See *route()* for details.

omni.web.routing.**put**(*template*, **arg*, ***kw*)
Decorates a function as a route, using PUT as method.

See *route()* for details.

omni.web.routing.**delete**(*template*, **arg*, ***kw*)
Decorates a function as a route, using DELETE as method.

See *route()* for details.

omni.web.routing.**patch**(*template*, **arg*, ***kw*)
Decorates a function as a route, using PATCH as method.

See *route()* for details.

omni.web.routing.**after**(*after_func*, **after_arg*, ***after_kw*)
Decorator that calls a function with the result of another.

Decorates a function and arranges for *after_func* to be called with the return value of the decorated function. Additional arguments can be passed to *after_func* by passing them to the decorator.

For example, the following can be used to automatically convert the values returned by the generate_data() into JSON:

```
>>> import json
>>> @after(json.dumps)
>>> def generate_data():
...     return { 'a': 42, 'b' }
...
>>> generate_data()
'{"a":42}'
```

Parameters

- **after_func** – Function to be run after the decorated function. It can be a callable object.
- **after_arg** – Additional arguments to be passed to *after_func*.
- **after_kw** – Additional keyword arguments to be passed to *after_func*.

omni.web.routing.**authenticate**(*get_realm*, *realm_param='realm'*)
Decorate a request handler to require HTTP basic authentication.

Typical usage:

```
class Authenticated(Dispatcher):
    def obtain_realm(self):
        return # ...

    @get("/realm/{r}/{page}")
    @authenticate(obtain_realm, "r")
    def handle_request(self, request, page):
        return get_page(page)
```

Parameters

- **get_realm** – Function or callable used to obtain a Realm instance. It will be passed the realm name as retrieved from the *realm_param* parameter of the original function. Pass None to disable inspecting the parameters of the decorated function.
- **realm_param** – Name of the parameter which contains the name of the realm to use. Note that the parameter will not be passed to the decorated function.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

O

A

D

G

M

O

P

R

U

V